

第三章 指令系统和寻址方式

- 3. 1 80X86寻址方式
- 3. 2 80X86机器语言指令概况
- 3. 3 80X86指令系统

本章目标

了解计算机的一般指令格式

掌握80X86CPU寻址方式

掌握80X86CPU指令系统

熟练掌握80X86CPU常用指令功能

计算机的一般指令格式

- ◆ **指令系统**：一组指令集。计算机所能执行的所有指令的集合就是指令系统
 - CPU依靠机器指令来计算和控制系统
 - 每款CPU在设计时就规定了一系列与其硬件电路相配合的指令集
- ◆ 指令系统决定了计算机能执行的全部基本操作。要使计算机完成一个特定的任务，就要告诉计算机按照怎样的顺序执行一个个基本操作（指令），这个具有约定顺序的一条条指令构成程序

常见的指令格式

指令：

操作码	操作数1	...	操作数n
-----	------	-----	------

零地址指令： RET

IRET

一地址指令： INC AX

DEC CX

二地址指令： MOV AX, [2000H]

ADD AH, BL

三地址指令： 很少使用

80X86最
常用

3.1 80X86寻址方式

寻址方式：指令指定操作数地址的方式

1、与数据有关的寻址方式

2、与转移地址有关的寻址方式

操作数通常保存在

(1) 指令中

MOV AX, 2000H

(2) CPU的寄存器中

MOV AX, BX

(3) 内存单元中

MOV AX, [2000H]

(4) I/O接口寄存器中

IN AH, 20H

3.1 80X86寻址方式

机器指令格式: OP dst, src

OP 操作码

src 源操作数

dst 目的操作数

例: MOV dst, src 把src送到dst

3.1.1 与数据有关的寻址方式

1. 立即寻址方式

立即寻址方式 —— 操作数在指令中给出

MOV AL, 5

MOV AX, 3064H

MOV EAX, 88663064H

指令

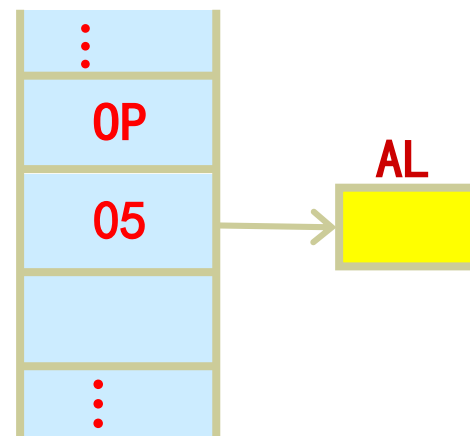
操作数

(1) 立即

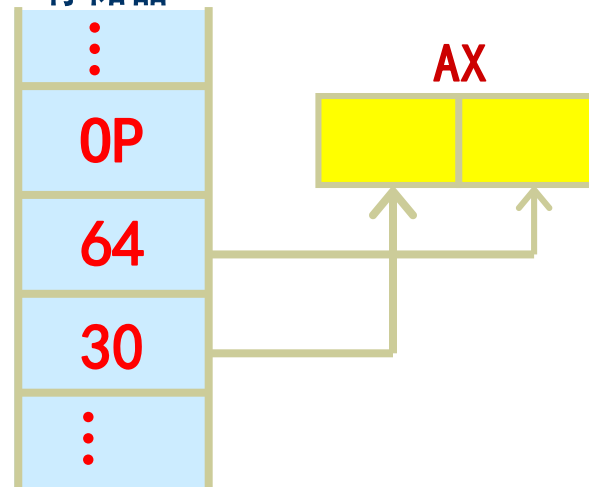
- * 经常用于给寄存器赋初值
- * 只能用于SRC字段
- * SRC 和 DST的字长一致

MOV AH, ~~3064H~~

存储器



存储器



2. 寄存器寻址方式

寄存器寻址方式* —— 操作数在指定的寄存器中

```
MOV  AL, BH
MOV  AX, BX
MOV  EAX, EBX
```



(2) 寄存器

- * 字节寄存器只有 AH AL BH BL CH CL DH DL
- * SRC 和 DST的字长一致

```
MOV  AH, BX
```

- * CS不能用MOV指令改变

```
MOV  CS, AX
```

```
例1 MOV  BX, AX      ;BX ← AX
例2 MOV  DI, 5678H   ;DI ← 5678H
例3 MOV  AL, 78H     ;AL ← 78H
例4 MOV  ECX, 7890ABCDH ;ECX ← 7890ABCDH
BX、AX、DI、AL、ECX均为寄存器寻址方式
```

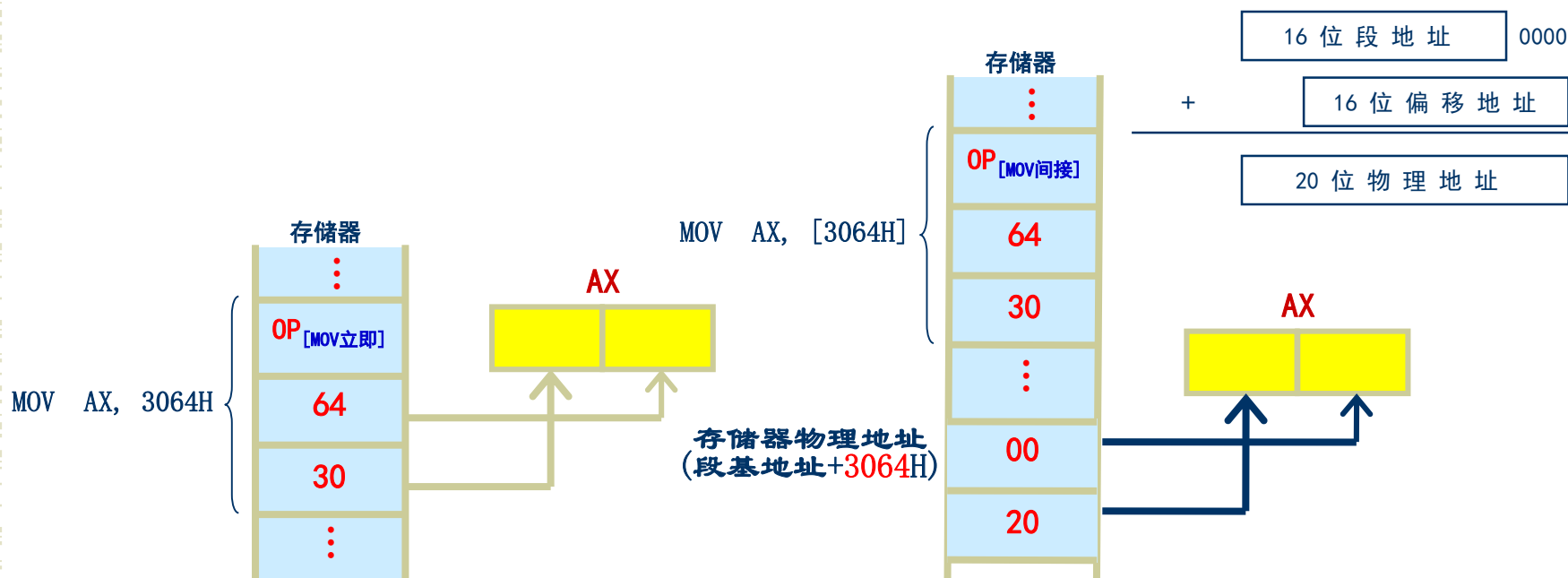
说明： 除立即寻址和寄存器寻址外，以下各种寻址方式的操作数都在除代码段以外的存储区中。

◆ **如何得到操作数呢？！**

- 通过不同寻址方式求得操作数在存储器中的地址，从而得到操作数

物理地址=段基址+偏移地址

- 如何指定操作数的段基址和偏移地址？



操作数的段基地址表示和生成

- ◆ 操作数的段基地址用段寄存器获得

- 若工作在实模式：段基址为段寄存器中的内容乘以16的值

段基地址= 段寄存器 × 16D

段基地址= 段寄存器 × 10H

- 若工作在保护模式：段基址通过段寄存器中的段选择子从描述符中得到

16 位 段 地 址 0000

+

16 位 偏 移 地 址

20 位 物 理 地 址

- ◆ 指令中给出的操作数逻辑地址格式为

段寄存器名：偏移地址

- 段寄存器名如何指定

- 段寄存器名可以缺省，这时使用默认约定的段寄存器
- 段寄存器名可以特别指定，用于跨越段访问，但也要遵循一定的规则

- 偏移地址如何指定

段基址和偏移量的约定情况

段寄存器名：偏移地址

操作类型	约定段寄存器	允许指定的段寄存器	偏移量
1. 指令	CS	无	IP
2. 堆栈操作	SS	无	SP
3. 普通变量	DS	ES、SS、CS	EA
4. 字符串指令的源串地址	DS	ES、SS、CS	SI
5. 字符串指令的目标串地址	ES	无	DI
6. BP用作基址寄存器	SS	DS、ES、CS	EA

数据操作

操作数的偏移地址 表示和生成

- ◆ 操作数的偏移地址又称为**有效地址EA**，所以下述各种寻址方式即为求得有效地址EA的不同途径以及在指令中如何表示
- ◆ 有效地址EA的四种成分：

基址、变址、比例因子、位移量

- (1) **位移量** (displacement) 是指令中指定的一个8位、16位或32位的数，它不是立即数而是一个地址的位移量

(2) **基址** (base) 是存放在基址寄存器中的内容

- 它是有效地址中的基址部分，通常用来指向数据段中数组或字符串的首地址

(3) **变址** (index) 存放在变址寄存器中的内容

- 它通常用来指定数组中的某个元素或字符串中的某个字符

(4) **比例因子** (scale factor) 是386及其后继机型新增加的寻址方式中的一个术语

- 其值可为1, 2, 4或8
- 在寻址中，可用变址寄存器的内容乘以比例因子来取得变址值
 - 这类寻址方式对访问元素长度1、2、4、8字节的数组特别有用

有效地址的计算

◆ 有效地址的用下式计算

$$EA = \text{基址} + (\text{变址} * \text{比例因子}) + \text{位移量}$$

在这4个成分中，除比例因子是固定值外，其他3个成分的数值都可正可负，以保证指针移动的灵活性。

◆ 注意：

- 8086/80286只能使用16位有效地址寻址
- 80386及其后继机型既可用32位有效地址寻址，也可用16位有效地址寻址

表3.1：16/32位有效地址 寻址时有4种成分组成

	16位寻址	32位寻址
位移量	0, 8位, 16位	0, 8位, 32位
基址寄存器	BX, BP	任何32位通用寄存器 (包括ESP)
变址寄存器	SI, DI	除ESP以外的32位通用 寄存器
比例因子	无	1, 2, 4, 8

$$EA = \text{基址} + (\text{变址} * \text{比例因子}) + \text{位移量}$$

指令中操作数的偏移地址 (有效地址) 表示

◆ 指令中操作数的有效地址表示形式：

$[\text{基址}] [\text{变址} * \text{比例因子}] [\text{位移量}]$

或者

$[\text{基址} + \text{变址} * \text{比例因子} + \text{位移量}]$

- 比例因子必须与变址一起组合使用
- 基址、变址、比例因子、位移量 4个成分有8种组合，即8种寻址方式

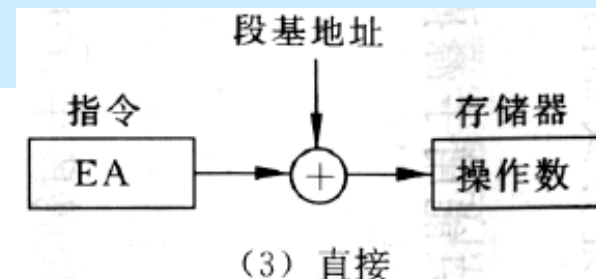
◆ 指令中操作数的地址表示形式：

段寄存器名：[基址+变址*比例因子+位移量]

[基址+变址*比例因子+位移量]，只有位移量情况

$$EA = [\text{位移量}]$$

3. 直接寻址方式



- ◆ 操作数存放在存储器的存储单元中，**有效地址EA**由指令直接给出

例：MOV BX, [2100H] ; DS: [2100H] → BX

- [] 内为操作数在段内的偏移地址
- 如果指令中缺省段寄存器名，直接寻址的段寄存器约定为数据段DS

物理地址计算公式： $PA = DS \times 16 + EA$

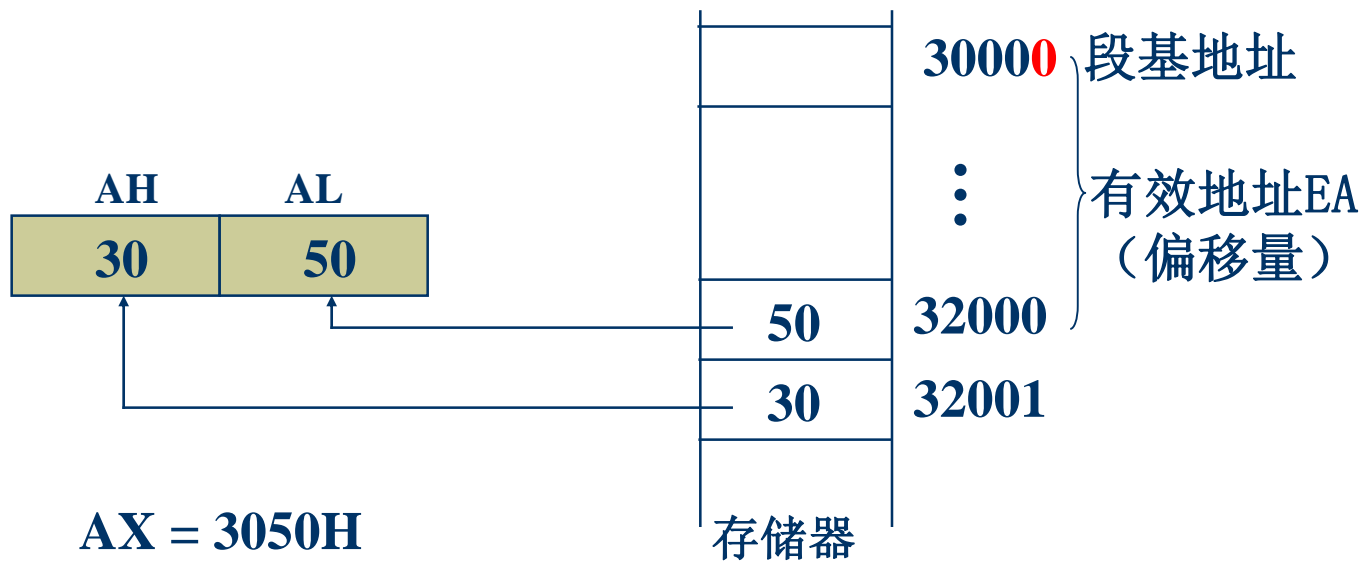
- 如果数据存放在其他段如附加数据段ES (代码段CS, 堆栈段SS)，偏移量XXXX处
 - 操作数地址表示为ES: [XXXX]
 - ◆ 对应的指令，如 MOV BX, ES: [100H]
 - 操作数物理地址计算时也要采用ES段

物理地址计算公式如下： $PA = ES \times 16 + EA$

例: MOV AX, [2000H]

EA=2000H, 假设DS=3000H

那么, $PA = DS \times 10H + EA = 30000 + 2000 = 32000H$



◆ 操作数地址可由符号地址表示

MOV AH, VALUE

VALUE的属性应该为地址

MOV AH, [VALUE]

- ◆ 两者等效，这时VALUE是操作数的符号地址，但必须在程序中提前定义属性和数值

◆ 使用变量时，要注意变量的属性

VALUE DB 10

× MOV AX, VALUE

✓ MOV AX, WORD PTR VALUE

- ◆ 实际上在汇编语言源程序中一般所看到的直接寻址方式都是用符号表示的，只有在DEBUG环境下，才有[2000H]这样的表示

[基址+变址*比例因子+位移量]，只有基址或变址情况

EA= [基址]或EA=[变址]

4. 寄存器间接寻址方式

- ◆ 操作数存放在存储器单元中
- ◆ **16位的操作数偏移地址EA**在指令指定的基址寄存器BX、BP或变址寄存器SI、DI中

■ **EA=指定寄存器内容**



(4) 寄存器间接

■ **物理地址的计算如下：**

$$PA = DS \times 16 + BX \quad (\text{或者SI、DI})$$

$$PA = SS \times 16 + BP$$

例：MOV CX, [SI] ; (DS × 16 + SI) → CX, 源操作数用SI
; 寄存器间接寻址方式

MOV CX, [BP] ; (SS × 16 + BP) → CX, 源操作数用BP
; 寄存器间接寻址方式

- ◆ **32位的操作数偏移地址**由指令指定的8个扩展寄存器指定。
EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

例. MOV AX, [BP] ; (SS:[BP]) → AX

- 若SS= 2000H, BP=0080H

存储单元 (20080H) = 12H

存储单元 (20081H) = 56H

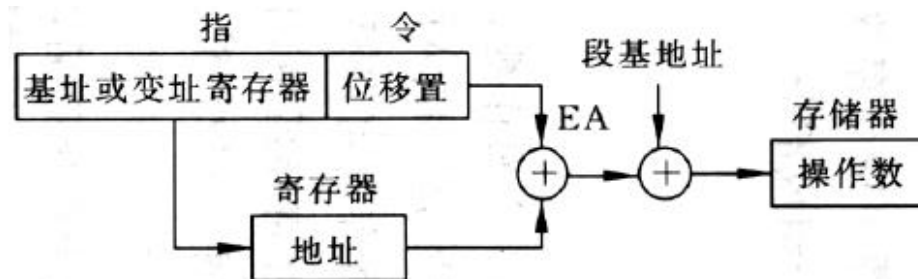
- 则物理地址 = $10H \times SS + BP$
= 20080H

- 该指令的执行结果是 AX= 5612H

[基址+变址*比例因子+位移量]，基址或变址+位移量情况
 $EA = [\text{基址} + \text{位移量}]$ 或 $EA = [\text{变址} + \text{位移量}]$

5. 寄存器相对寻址方式

- ◆ 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的寄存器BX、BP、SI、DI和指令中给定的位移量相加得到



- ◆ 物理地址计算：

$$PA = DS \times 16 + \left\{ \begin{array}{c} BX \\ SI \\ DI \end{array} \right\} + \text{位移量} \quad (5) \text{ 寄存器相对}$$

$$PA = SS \times 16 + BP + \text{位移量}$$

- ◆ 例：

- `MOV AX, NUM [BX]` ; $(DS \times 16 + \text{BX} + \text{NUM}) \rightarrow AX$
- `MOV BX, CCC [BP]` ; $(SS \times 16 + \text{BP} + \text{CCC}) \rightarrow BX$

**例： MOV AX, COUNT[SI]
或 MOV AX, [COUNT+SI]**

◆ **假设： DS=3000H, SI=2000H, COUNT=3000H**

(35000H)=1234H

◆ **那么： EA = SI+COUNT=2000+3000 = 5000H**

PA = 30000+5000 = 35000H

◆ **则 AX=1234H**

[基址+变址*比例因子+位移量]，同时有基址和变址情况
 $EA = [基址 + 变址]$

6. 基址变址寻址方式

- 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的基址寄存器（BX, BP）和变址寄存器（SI, DI）内容相加



(6) 基址变址

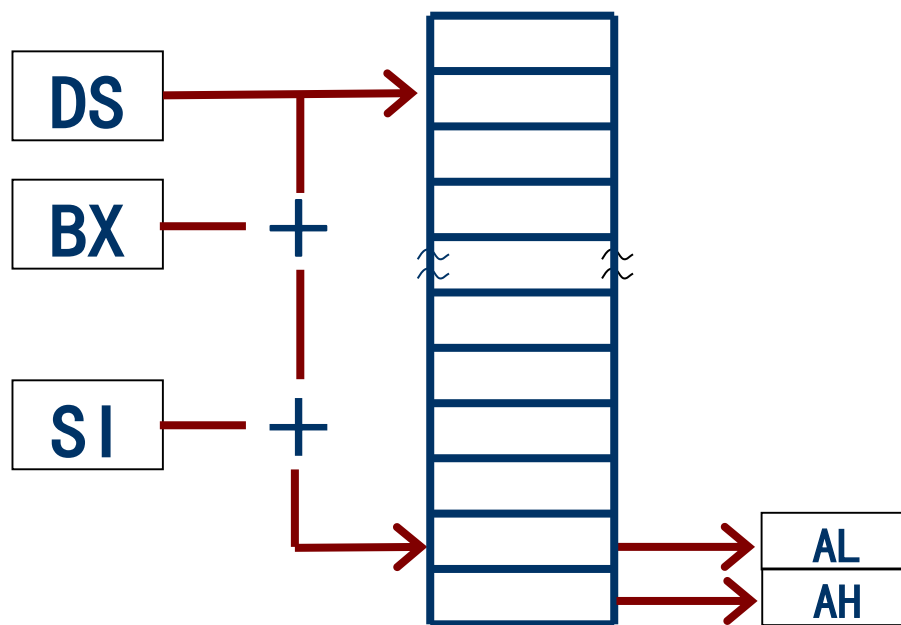
- 物理地址计算：

$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix}$$

$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix}$$

- 例：MOV AX, [BX][SI] 或 MOV AX, [BX+SI]
($DS \times 16 + BX + SI$) \rightarrow AX

基址变址寻址方式执行情况

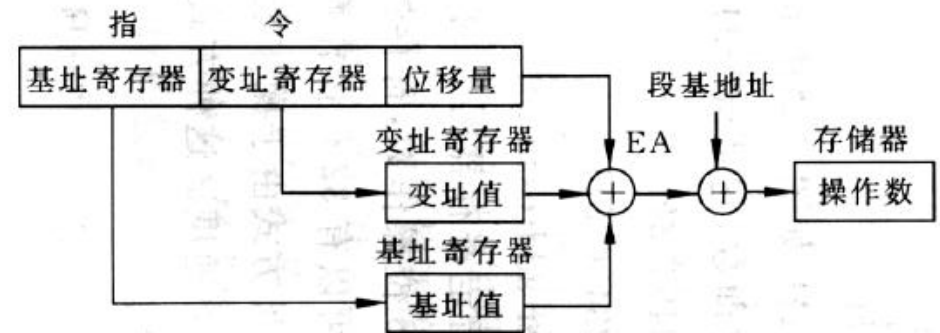


[基址+变址*比例因子+位移量]，同时有基址、变址、位移量情况

$$EA = [\text{基址} + \text{变址} + \text{位移量}]$$

7. 相对基址变址寻址方式

- 操作数存放在存储器的单元中，操作数的偏移地址由指令指定的基址寄存器（BX、BP）和变址寄存器（SI、DI）以及位移量相加



- 物理地址计算：

$$PA = DS \times 16 + BX + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \text{位移量} \quad (7) \text{ 相对基址变址}$$

$$PA = SS \times 16 + BP + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \text{位移量}$$

MOV AX, MASK[BX][SI], MOV AX, MASK[BX+SI],
MOV AX, [MASK+BX+SI]等同

寻址方式总结

(1) 3、4、5、6、7这五种寻址方式

- 所指定的操作数都存在于内存单元中
- 处理器根据指令中给出的地址信息求出存放操作数的内存单元偏移地址(或有效地址)
 - 计算机根据指令给出的寻址方式求出操作数的偏移地址就是求出了操作数地址的段内偏移地址
- 再加上段基地址, 得到操作数在内存中的物理地址
- 然后对存放在内存的操作数进行存取操作

寻址方式总结

(2) 4、5、6、7这四种寻址方式

- 使用了BX、BP、SI、DI寄存器
- 只要指令寻址时使用了BP，计算物理地址时约定段是SS段
- 指令寻址时使用了除BP以外的其它寄存器，计算物理地址时约定段为DS段

(3) 若操作数部分使用了段超越（即段更换前缀），则计算物理地址时使用超越段

- 如 MOV AX, ES: [100H]，计算物理地址时段地址为ES，这条指令的源操作数的物理地址= $ES \times 16 + 100H$

[基址+变址*比例因子+位移量]，同时有变址、比例因子、位移量情况

$$EA = [\text{变址} * \text{比例因子} + \text{位移量}]$$

8. 比例变址寻址方式

- ◆ 相对比例变址寻址方式更好理解
- ◆ 32位地址寻址方式，80386以上微处理器
- ◆ 操作数的有效地址是变址寄存器的内容乘以指令中指定的比例因子，再加上位移量

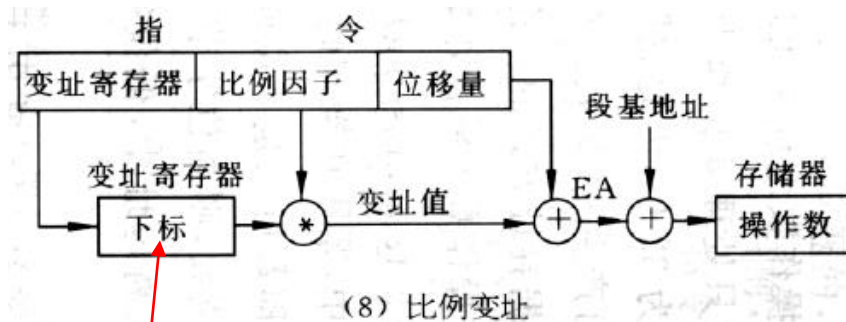
$$EA = [\text{变址} * \text{比例因子} + \text{位移量}]$$

$$PA = DS \times 16 + \left\{ \begin{matrix} SI \\ DI \end{matrix} \right\} * S + \text{位移量}$$

$$PA = SS \times 16 + \left\{ \begin{matrix} SI \\ DI \end{matrix} \right\} * S + \text{位移量}$$

例如：MOV EAX, NUM[ESI × 8]

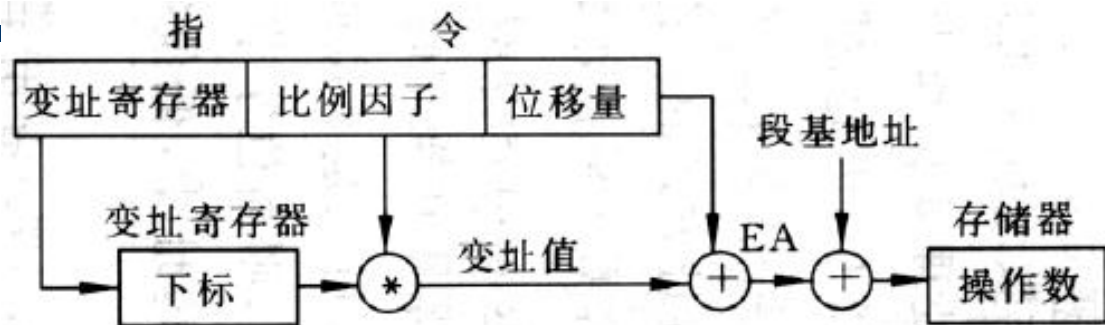
$$PA = DS \times 10H + ESI \times 8 + \text{NUM}$$



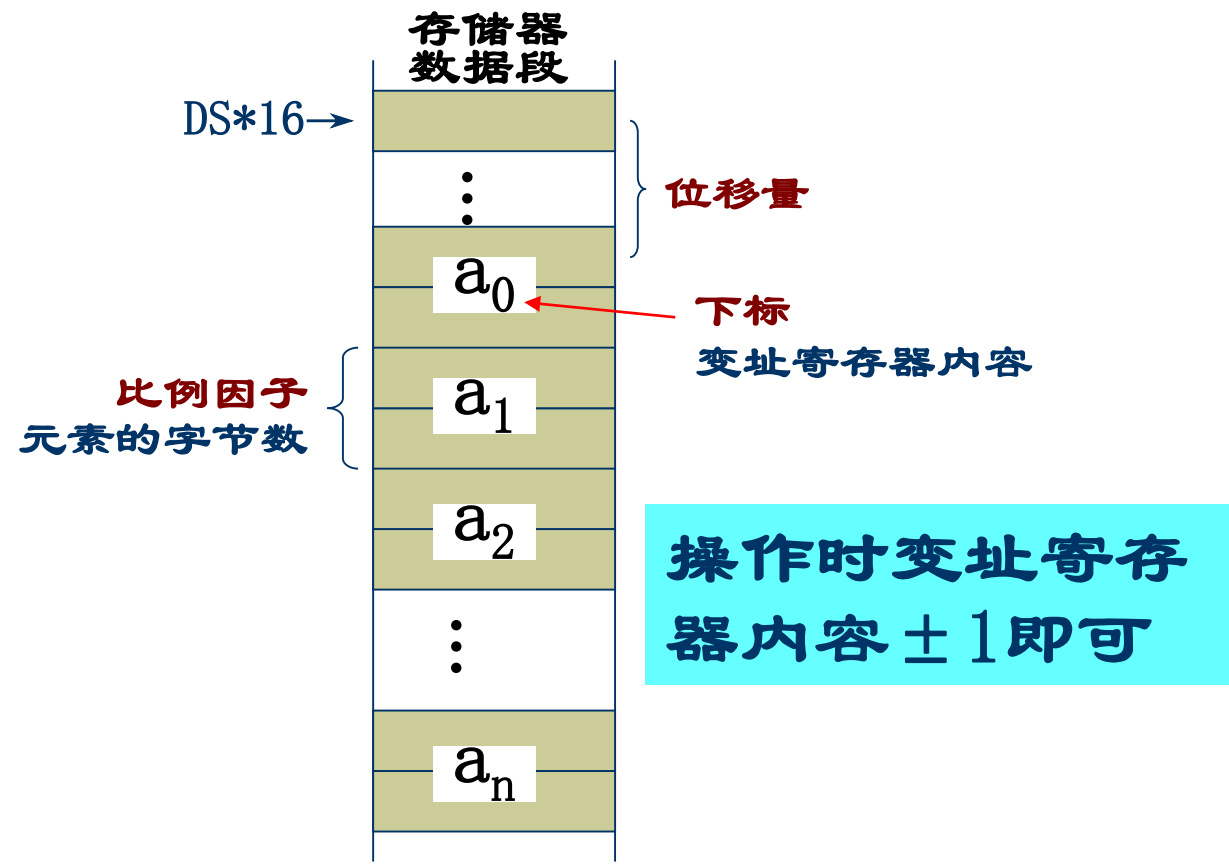
?

下标一定从0开始

字符串和数组处理使用



(8) 比例变址



[基址+变址*比例因子+位移量]，同时有基址、变址、比例因子情况

$$EA = [\text{基址} + \text{变址} * \text{比例因子}]$$

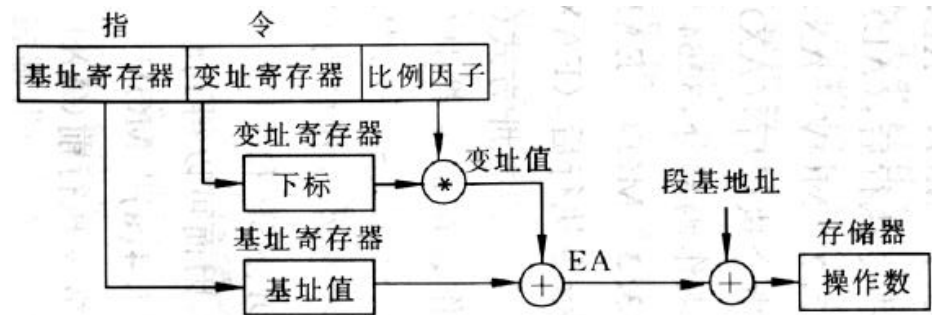
9. 基址比例变址寻址方式

- ◆ 32位地址寻址方式，80386以上微处理器
- ◆ 操作数的有效地址是变址寄存器的内容乘以比例因子再加上基址寄存器的内容

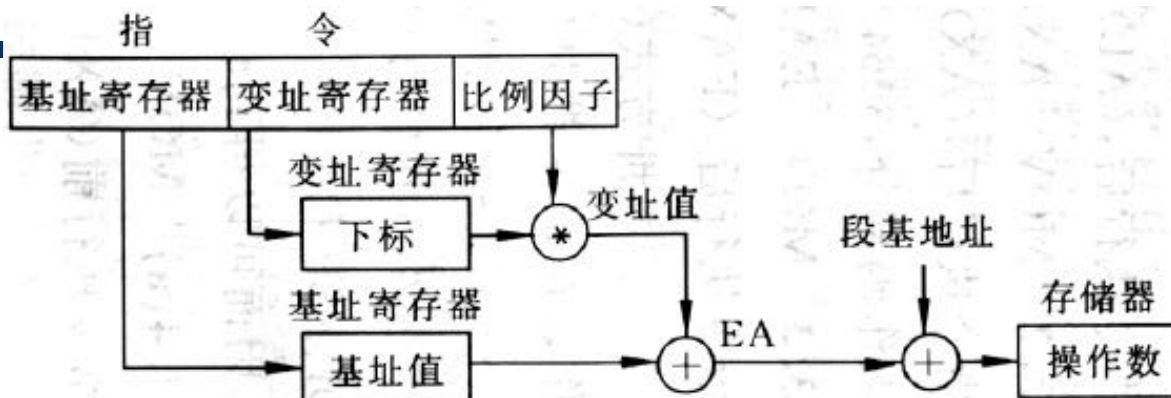
$$EA = [\text{基址} + \text{变址} * \text{比例因子}]$$

$$PA = DS \times 16 + BX + \begin{cases} SI \\ DI \end{cases} * S$$

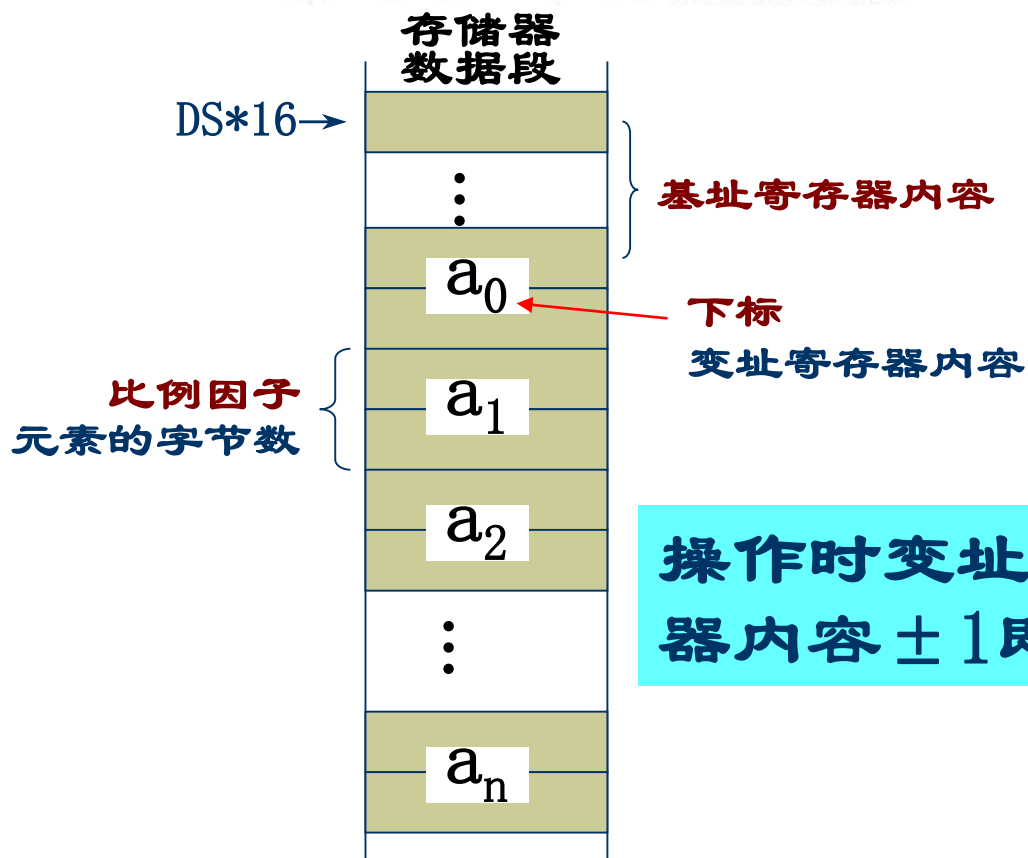
$$PA = SS \times 16 + BP + \begin{cases} SI \\ DI \end{cases} * S$$



(9) 基址比例变址



(9) 基址比例变址



基址寄存器可根据数组起始地址在程序中设置，编程灵活方便

操作时变址寄存器内容 ± 1 即可

[基址+变址*比例因子+位移量]

EA=[基址+变址*比例因子+位移量]

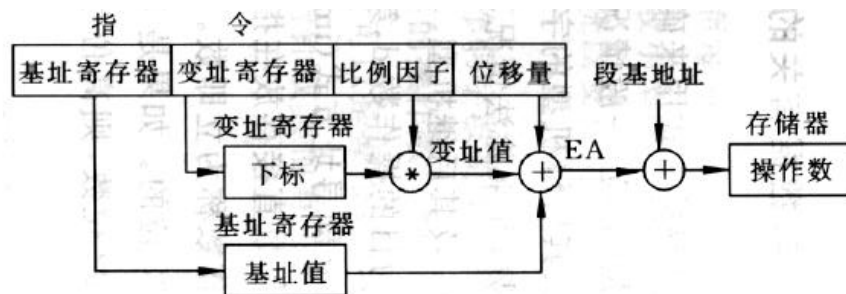
10. 相对基址比例变址寻址方式

- ◆ 32位地址寻址方式，80386以上微处理器
- ◆ 操作数的有效地址是变址寄存器的内容乘以比例因子，加上基址寄存器的内容再加上位移量

EA=[基址+变址*比例因子+位移量]

$$PA = DS \times 16 + BX + \left\{ \begin{matrix} SI \\ DI \end{matrix} \right\} * S + \text{位移量}$$

$$PA = SS \times 16 + BP + \left\{ \begin{matrix} SI \\ DI \end{matrix} \right\} * S + \text{位移量}$$



例：MOV AX, NUM [BX][SI*2]

完成 $(DS \times 16 + BX + SI \times 2 + \text{OFFSET NUM}) \rightarrow AX$

也可写成：MOV AX, [BX + SI*2 + NUM]

多个字符串和
数组处理使用

3.1.2 与转移地址有关的寻址方式

◆ 改变程序执行顺序的指令

- 若需要改变程序的正常执行顺序，转移到所要求的指令处执行程序时，可以安排一条**转移指令**或**转子指令**（CALL指令）

◆ 这种寻址方式主要是如何确定转移指令及CALL指令的**转向地址**

- 程序指令地址是由代码段寄存器CS和指令指针IP决定

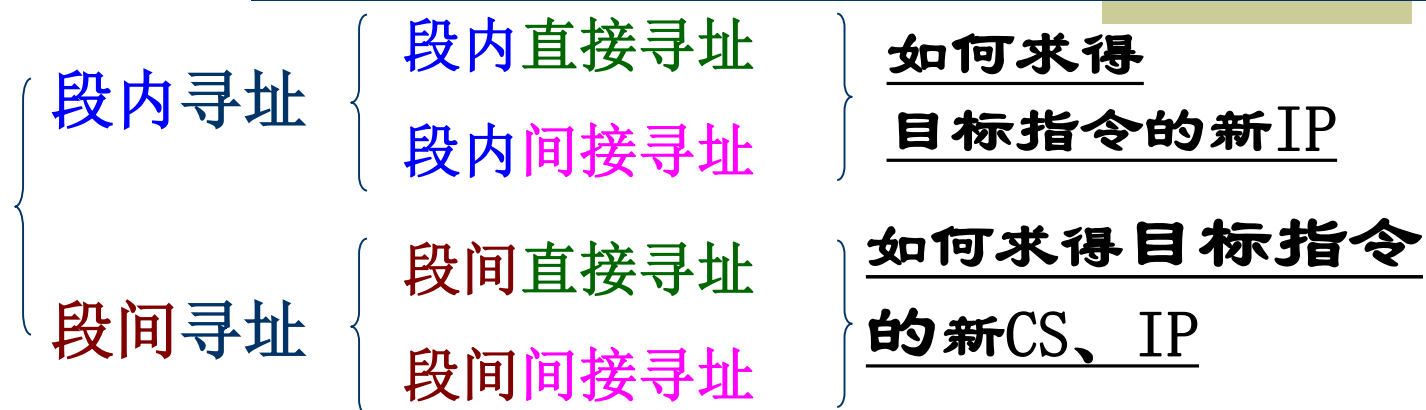
$$CS*16d+IP$$

- 功能就是修改CS和IP的值, 确定目的地址

◆ 转移的目的地址位置

- 可以段内转移（转移指令和转移的目的地址在同一代码段内）
- 也可以段间转移（转移指令和转移的目的地址不在同一代码段）
- 与转移地址有关的有四种寻址方式

3.1.2 与转移地址有关的寻址方式



段内寻址：转移指令与转向的目标指令在同一代码段中
CS内容不变，IP内容修改

段间寻址：转移指令与转向的目标指令在两个代码段中
CS和IP内容修改

直接寻址：转向的目标指令地址由转移指令直接指明

段间寻址：转向的目标指令地址由转移指令中的寄存器或存储单元内容给出
以无条件转移为例介绍

1. 段内直接寻址

- ◆ 转向的有效地址EA是当前IP寄存器的内容和指令中指定的8位或16位 位移量之和。如

```
code1 segment
...
IP当前 → jmp skip ; skip 为转向的目标地址
...      { skip数值=skip符号地址的有效地址-当前IP寄存器内容
...      { JMP指令执行后,  $IP_{新} = IP_{当前} + skip$ 
skip: nop
...
code1 ends
```

当程序执行完jmp skip指令后,就跳转到skip所指的nop指令执行。jmp和nop指令之间的指令不被执行。

转向的有效地址EA =

$$\begin{array}{c} \text{IP}_{\text{当前}} \\ + \\ \text{位移量 (8bit / 16bit)} \end{array}$$



IP_新

新指令物理地址 = $\text{CS} \times 16 + \text{IP}_{\text{新}}$

例:



- 位移量为16bit时, **近转移**:

JMP NEAR PTR NEXT , 位移量范围: $-32768 \sim +32767$

- 位移量为8bit时, **短转移**:

JMP SHORT NEXT , 位移量范围: $-128 \sim +127$

段内直接寻址示例

转向的有效地址 = 当前(IP) + 位移量(8bit/16bit)

```
JMP     SHORT NEXT
MOV     AX, 0
MOV     BX, 0
MOV     CX, 0
NEXT:   ret
```

12A2:0005	EB09	JMP	0010
12A2:0007	B80000	MOV	AX, 0000
12A2:000A	BB0000	MOV	BX, 0000
12A2:000D	B90000	MOV	CX, 0000
12A2:0010	CB	RET	F

2. 段内间接寻址

- ◆ 转向的目标指令的有效地址EA，即新IP内容是在寄存器或存储单元的内容
 - 跳转指令中给出寄存器名或存储单元的有效地址
 - 用数据寻址方式中除立即数以外的任何一种寻址方式
 - 取得转向的目标指令的有效地址
 - 用所得的目标指令的有效地址EA取代IP寄存器的内容即可实现段间跳转
- ◆ 新指令物理地址 = $CS \times 16 + IP_{\text{新}}$
- ◆ 例如 `jmp si`

Offset 告诉汇编程序取相对于段基地址的偏移量

```
code1    segment
          assume  cs:code1
start:    mov     si, offset next    ; next位置的偏移地址送SI
          jmp     si                ; 转向next, 寄存器间接寻址
          mov     ax, 1000h
          mov     bx, 2001h
next:     nop                      ; next是符号地址（或称标号）
          ...
code1     ends
```

要想转到next处，只要将IP内容设置为next处的偏移量
CPU就会从next处读取指令并执行

- ◆ 当程序执行完`mov si, offset next`指令后，`si`中得到了next标号在代码段中的偏移量
- ◆ 执行`jmp si`指令后，`SI`内容送`IP`，程序就跳转到next所指的`nop`指令执行
- ◆ `jmp si`和`next: nop`指令之间的两条`mov`指令不被执行

段内间接寻址得到的有效地址是16位，属于近转移

例： BX=1256H, SI=528EH, TABLE=20A2H
DS=2000H, (232F8H)=3280H, (264E4H)=2450H

JMP BX ; BX → IP, IP_新=1256H

实模式下，寄存器寻址只能使用16位寄存器

JMP TABLE[BX]
JMP *WORD PTR* TABLE[BX] } ; IP_新=3280H

$DS*10H+BX+TABLE=20000+1256+20A2=232F8H$ (232F8H) → IP

JMP [BX][SI]
JMP *WORD PTR* [BX][SI] } ; IP_新=2450H

$DS*10H+BX+SI=20000+1256+528E=264E4H$ (264E4H) → IP

3. 段间直接寻址

- ◆ 指令中直接提供了转向的段地址和偏移地址
 - 用指令中提供的转向段地址和偏移地址取代CS和IP
 - 新指令物理地址 = $CS_{\text{新}} \times 16d + IP_{\text{新}}$
- ◆ 段间直接寻址转移指令

`jmp far ptr seg2`

- 其中seg2为转向的目标指令的符号地址
- far ptr为段间转移的伪操作符
- 汇编语言程序编写时用符号地址（标号）
- 如果不用符号地址时，指令中直接给出段基地址：偏移地址

； 定义第一个代码段

```
code1    segment
```

```
begin:    ...
```

```
    jmp far ptr next ;转向next去执行,next为另一个代段的符号地址
```

```
    ...
```

```
code1    ends
```

；第一个代码段结束

汇编程序会生成机器指令

JMP CS':IP'

； 定义第二个代码段

```
code2    segment
```

```
    ...
```

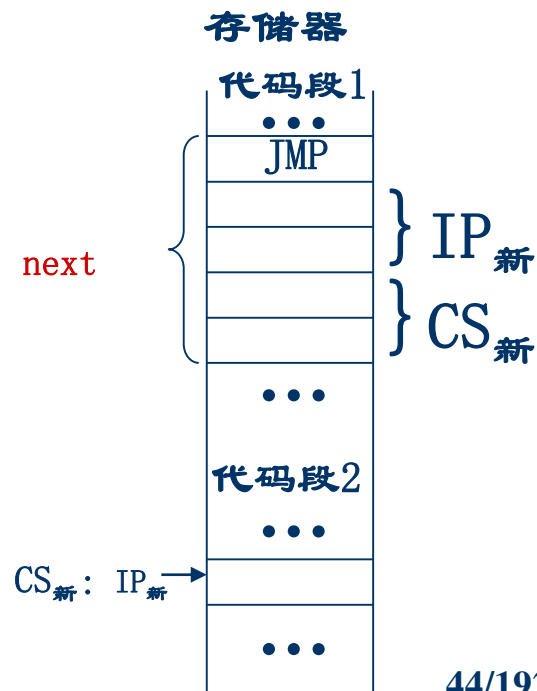
```
next:    ...
```

```
    ...
```

```
code2    ends
```

；第二个代码段结束

```
    jmp far ptr next
```

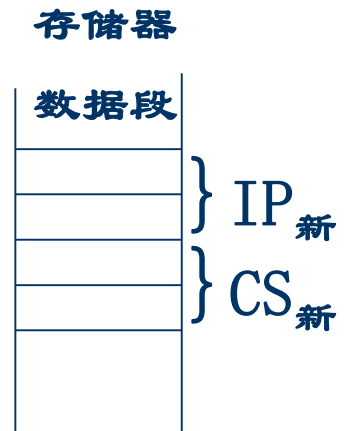


4. 段间间接寻址

◆ 用存储器中的两个相继字内容来取代IP和CS寄存器中的当前内容

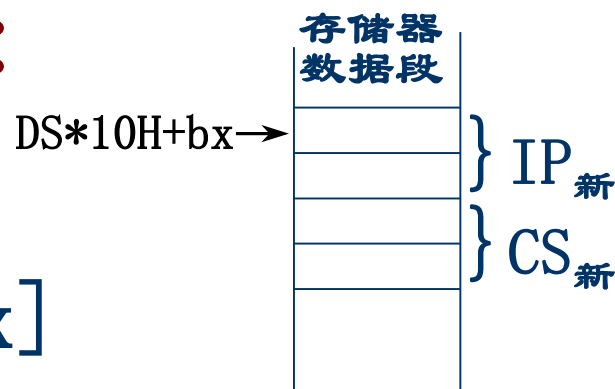
- 指令中给出这两个相继字的存储器寻址的有效地址
- 存储单元中的内容可以用数据寻址方式中除立即数和寄存器以外的任何一种寻址方式取得
- 设置新的CS、IP内容

◆ 新指令物理地址 = $CS_{\text{新}} \times 16 + IP_{\text{新}}$



段间间接寻址指令举例：

`jmp dword ptr [bx]`



- 其中 $[bx]$ 说明获取目标指令地址的寻址方式为寄存器间接寻址方式
- `dword ptr` 为双字操作符，说明转向地址需取双字，为段间转移指令
- 执行结果：
 $(DS*10H+BX+1, DS*10H+BX) \rightarrow IP$
 $(DS*10H+BX+3, DS*10H+BX+2) \rightarrow CS$

```

data1 segment
addrtab dw offset seg2
        dw seg seg2
data1 ends

```

addrtab

存储器

Seg2处
偏移地址

Seg2处
段基地址

Data1
数据段

```

code1 segment
begin:  ...
        mov  bx, offset addrtab
        jmp  dword ptr [bx]
        ...
code1 ends

```

; 转移目标地址的偏移地址的地址送
 ; [bx] → IP, [bx+2] → CS, 间接寻址到
 ; 另一个代码段

; 转移的目标代码段

```

code2 segment
        ...
    seg2:  ...
        ...
code2 ends

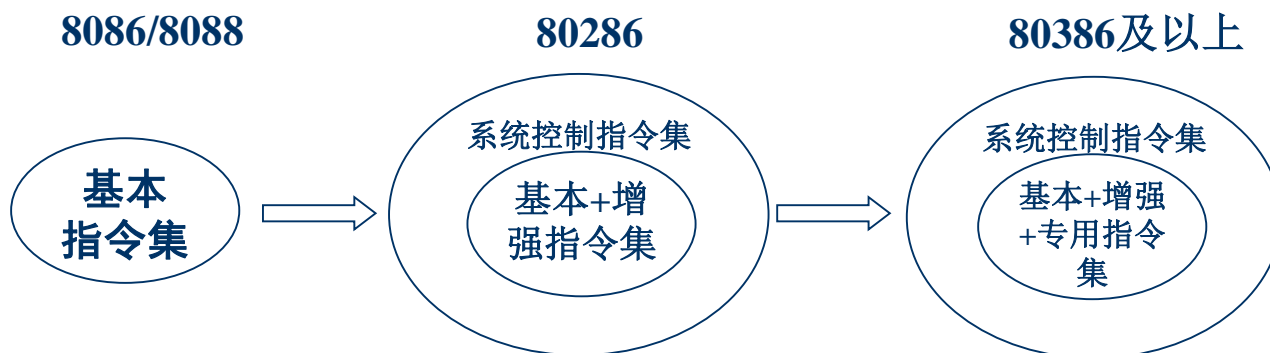
```



注意：

- ◆ 条件转移指令只能使用段内直接寻址方式
- ◆ 无条件转移（JMP）和转子指令（CALL）可用四种方式的任何一种

3.2 80X86机器语言指令概况



Intel 80x86指令集的发展

- ◆ 如果不基于汇编语言编程，一定要仔细阅读
- ◆ 如果想设计优化的汇编语言，也一定要仔细阅读

3.2.1 操作码的机器语言表示

◆ 8086机器语言指令操作码

■ 多字节指令

■ 一条指令1-7个字节

■ 机器码长度8-11位，一般情况下8位

■ 多数指令操作码格式：

操作码

mod-reg-r/m

位移量

立即数

1-2字节

0-1字节

0-2字节

0-2字节



- w=1, 字操作；w=0, 字节操作

- d双操作数有效；**操作数最多只能有一个放在存储器中**；

d=1, 寄存器用于目的操作数；d=0, 寄存器用于源操作数

■ 立即数寻址方式时，操作码格式：



- s=1, 按符号扩展规则将立即数从8位扩展为16位

- sw=00, 字节操作；sw=01, 有16位立即数且字操作

- sw=10, 无意义；sw=11, 8位立即数，但需字操作

3.2.2 寻址方式的机器语言表示

操作码	mod-reg-r/m	位移量	立即数
1-2字节	0-1字节	0-2字节	0-2字节

◆ 8086机器语言指令中寻址方式表示

- 一般是指令的第二个字节

- 寻址方式字节表示：

mod	reg	r/m
-----	-----	-----

- reg与操作码中w结合使用指定寄存器

- ◆ P50, 见3.3表

OP	d	w
----	---	---

- mod, r/m结合在一起确定寻址方式

- ◆ P51, 见3.4表

- 指定段跨越前缀表示：

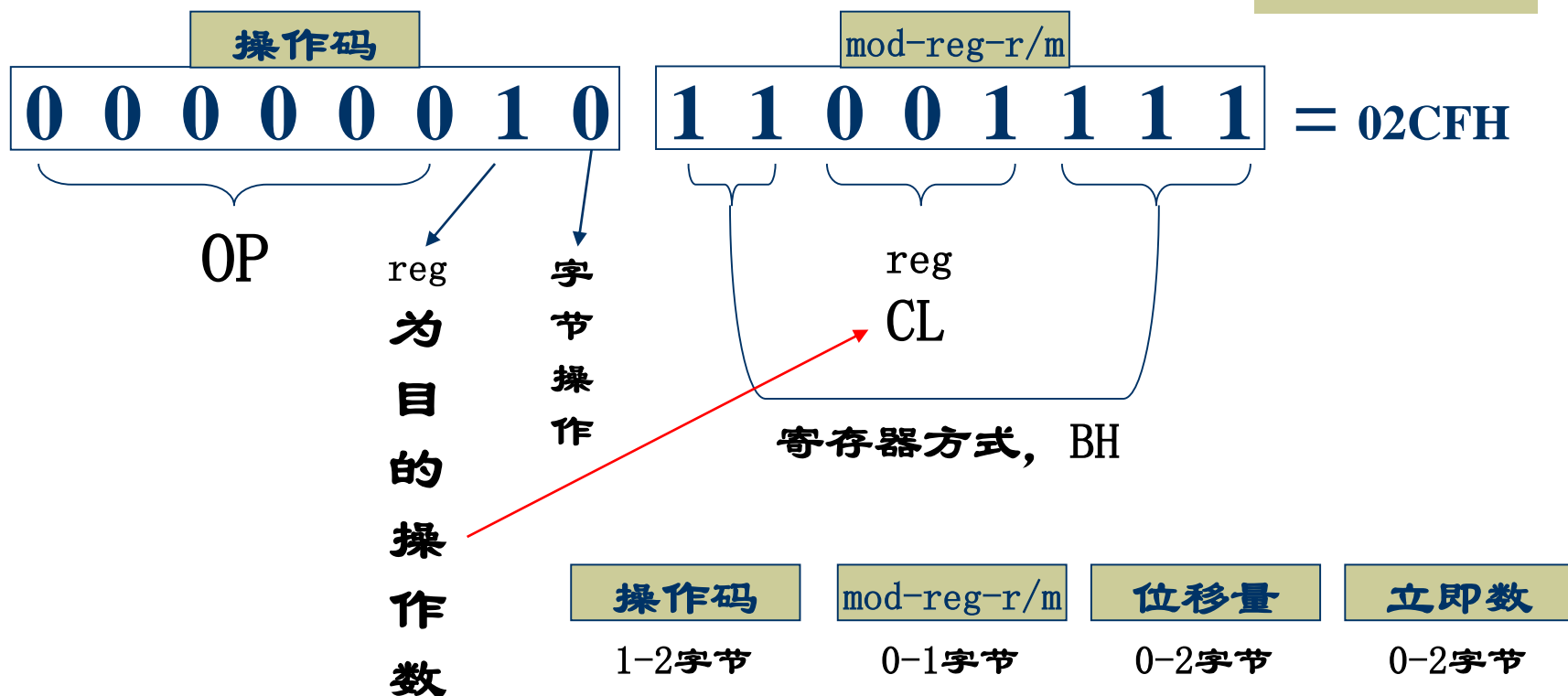
001	SEG	110
-----	-----	-----

- 放在指令之前的一个字节

- 001, 110段前缀标志

- SEG指定4个段寄存器中的一个, 见表3.5

例如: ADD CL, BH



结果: CL+BH → CL

汇编指令： ADD CL, BH

由汇编程序
将汇编指令
翻译成
二进制代码的机器指令

- 如果没有汇编程序就需要人工差错、查表翻译等
- 将人工查表翻译的处理过程编成程序，这就是汇编程序

机器指令： 0 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 = 02CFH

OP
ADD

reg
为
目的
操作
数

字节操作

reg
CL

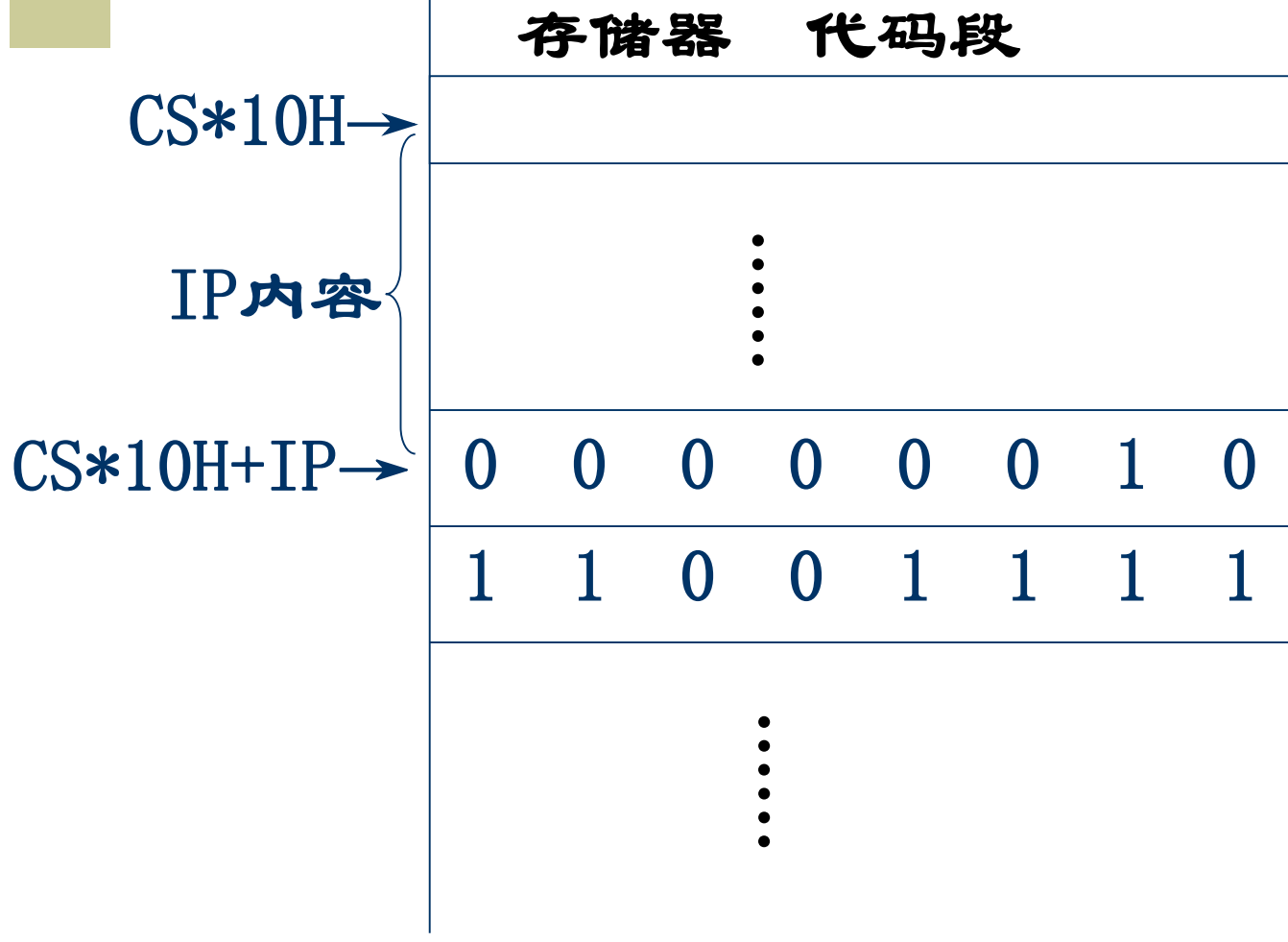
寄存器方式，BH

结果：CL+BH→CL

注意汇编程序
和汇编源
程序概念

ADD CL, BH

0 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 = 02CFH



3.2.3 指令的执行时间

◆ 指令执行时间

- 指令执行时间=取指令+取操作数+执行操作+传送结果
- 每一步至少一个时钟周期

◆ 指令执行时间用时钟周期数表示

◆ 请见P56，表3.6、3.7、3.8

- 表中为单条指令执行时间
- 现代处理器中由于流水线、超标量等，**一组指令（程序）的总执行时间**会大大缩短，每条指令的平均执行时间<1个时钟周期
- 编程时尽量使用时钟周期数少、指令字节数少的指令实现相同的功能

3.2.4 机器指令格式

◆ 8086/80286 指令格式

操作码	mod-reg-r/m	位移量	立即数
1-2字节	0-1字节	0-2字节	0-2字节

- 16位地址, 16位偏移量; 8、16位数据

◆ 80386以上 指令格式

地址长度	操作数长度	操作码	mod-reg-r/m	比例-变址	位移量	立即数
0-1字节	0-1字节	1-2字节	0-1字节	0-1字节	0-4字节	0-4字节

- 32位地址/ 32位地址, 16、32位偏移量;
- 段前缀; 8、16、32位数据

3.3 80X86的指令系统

3.3.1 数据传送指令

3.3.2 算术指令

3.3.3 逻辑指令

3.3.4 串处理指令

3.3.5 控制转移指令

3.3.6 处理机控制指令

学习时注意：

1. 指令的基本功能
2. 指令的执行对标志位的影响
3. 对寄存器和存储单元内容的影响
4. 对寻址方式或寄存器使用的限制和隐含使用的情况

3.3.1 数据传送指令

- ◆ 数据传送指令负责把数据、地址或立即数传送到寄存器或存储单元中
- ◆ 这类指令可实现：
 1. 寄存器之间 的数据交换
 2. 寄存器和存储器之间 的数据交换
 3. 立即数送 寄存器/存储器 单元中
 4. 把地址送 指定的寄存器
 5. 寄存器/存储器单元的内容 压入堆栈，或弹出堆栈
 6. 累加器和I/O端口之间 的数据传送
 7. 标志寄存器和AH之间 的数据交换

3.3.1 数据传送指令

这类指令分为如下五类：

- 1、通用数据传送指令
- 2、累加器专用传送指令
- 3、地址传送指令
- 4、标志寄存器传送指令
- 5、类型转换指令

1、通用数据传送指令

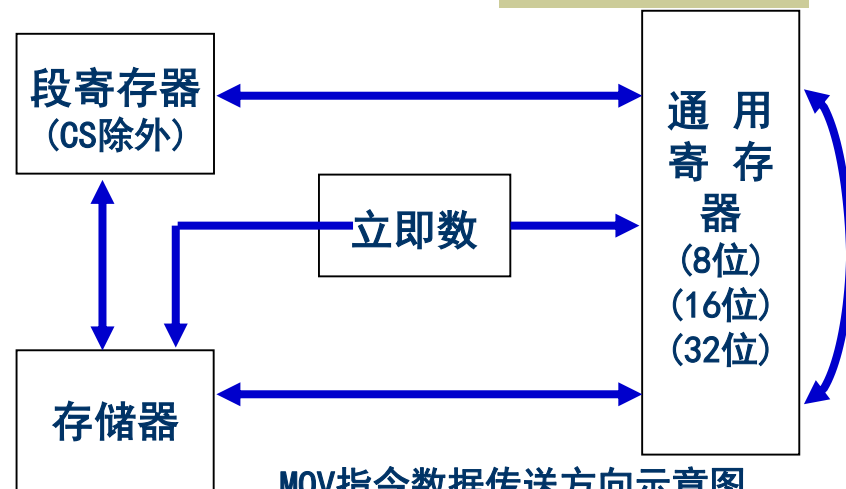
- ① **MOV** (move)
- ② **MOVSX** (move with **sign**-extend) (386及后继机型)
- ③ **MOVZX** (move with **zero**-extend) (386及后继机型)

- ④ **PUSH** (push onto the stack)
- ⑤ **POP** (pop from the stack)
- ⑥ **PUSHA/PUSHAD** (push all(16/32) registers)
- ⑦ **POPA/POPAD** (pop all(16/32) registers)
- ⑧ **XCHG** (exchange)

① MOV指令

传送指令: `MOV DST, SRC`

执行操作: $(DST) \leftarrow (SRC)$



注意:

- 立即数不能作为目标操作数
- 立即数不能直接送段寄存器 × `MOV DS, 2000H`
- DST不能是CS, 因为随意修改CS会引起不可预料的结果
- DST、SRC不同时为段寄存器 × `MOV DS, ES`
- 两个存储单元之间不能直接传送
- 不影响标志位

例: **MOV AX, DATA_SEG**
MOV DS, AX

段名

这里按段基址处理

例: **MOV AL, 'E'** ; **MOV AL, 45H**

例: **MOV BX, OFFSET TABLE**

例: **MOV AX, Y[BP][SI]**

② MOVZX带符号扩展传送指令

- ◆ 386及后继机型可用
- ◆ **格式：MOVZX DST, SRC**
- ◆ **功能：(DST) ← 符号扩展 (SRC), DST空出的位用SRC的符号位填充**
 - 该指令的源操作数SRC可以是8位(16位)的寄存器或存储单元的内容, 不能是立即数
 - 目的操作数DST则必须是16位(32位)的寄存器, 传送时把源操作数符号扩展送入目的寄存器
 - MOVZX不影响标志位



MOV DL, 80H

MOVSX AX, DL

AX中得到80H的带符号扩展值FF80H

MOV VAR, 56H

MOVSX AX, VAR

AX中得到56H的带符号扩展值0056H

③ MOVZX 带零扩展传送指令

- ◆ 386及后继机型可用
- ◆ 格式: MOVZX DST, SRC
- ◆ 功能: (DST) ← 零扩展 (SRC), DST空出的位用0填充
 - 该指令的源操作数和目的操作数以及对标志位的影响均与MOVSX指令相同
- ◆ 区别: MOVSX的源操作数为带符号数扩展
MOVZX的源操作数为无符号数扩展

MOV DL, 80H

MOVZX AX, DL ; AX= 0080H

MOVZX EAX, DL ; EAX= 00000080H

MOV CX, 1234H

MOVZX EAX, CX ; EAX= 00001234H

MOV VAR, 56H

MOVZX AX, VAR ; AX=0056H

- **使用MOVSX指令可以方便地实现对带符号数的扩展**
- **使用MOVZX指令可以方便地实现对无符号数的扩展**

④ PUSH指令

⑤ POP指令

进栈指令: **PUSH SRC**

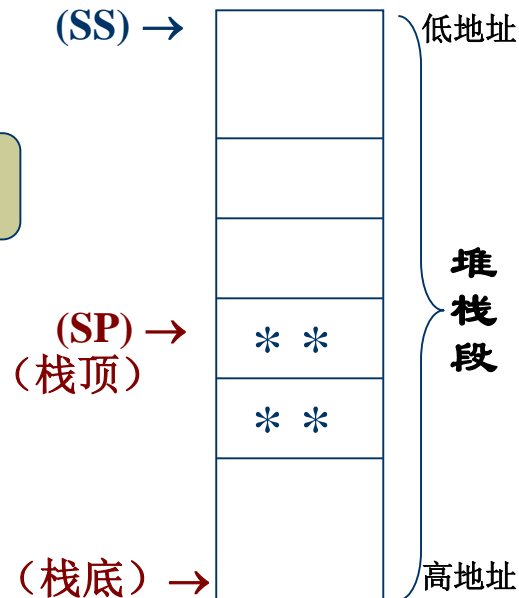
执行操作: $(SP) \leftarrow (SP) - 2$
 $((SP)+1, (SP)) \leftarrow (SRC)$

目标地址默认为
堆栈栈顶

出栈指令: **POP DST**

执行操作: $(DST) \leftarrow ((SP)+1, (SP))$
 $(SP) \leftarrow (SP) + 2$

源操作数地址默
认为堆栈栈顶



8086 PUSH/POP指令格式

PUSH/POP REG

PUSH/POP MEM

PUSH/POP SEGREG

不允许立即数操作

80286以上PUSH/POP指令格式

PUSH/POP REG

PUSH/POP MEM

PUSH/POP DATA

PUSH/POP SEGREG

例： 假设 $AX = 2107\text{ H}$, 执行 `PUSH AX`



`PUSH AX` 执行前

`PUSH AX` 执行后

堆栈指针SP内容自动修改，并且是地址减量

例： POP BX



POP BX 执行前

POP BX 执行后

BX = 2107H

堆栈指针SP内容自动修改，并且是地址增量

- ◆ 堆栈：“后进先出”的存储区，存在于堆栈段中，SP 在任何时候都指向栈顶
- ◆ 若SRC是16位操作数，则堆栈指针寄存器减2；若SRC是32位操作数，则堆栈指针寄存器减4
- ◆ 若DST是16位，则堆栈指针寄存器加2；若DST是32位，则堆栈指针寄存器加4

注意：

- * 堆栈操作必须以字为单位

- * 不影响标志位

- * 8086不能用立即寻址方式 × PUSH 1234H

- * DST不能是CS × POP CS

CS不能人为改动，需要时可以借助RET指令让CPU自己改动

- * PUSH、POP指令一般配对使用

例: **PUSH AX**
 PUSH BX
 ...
 ...
 POP BX
 POP AX

- ◆ 堆栈主要用于对现场数据的保护与恢复、子程序与中断服务返回地址的保护与恢复、参数传递等

⑥ **PUSHA/PUSHAD** 所有 (16/32位 P62) 寄存器进栈指令

PUSHA: 8个16位通用寄存器依次进栈 (AX、BX、CX、DX, 指令执行前的SP、BP、SI、DI)

PUSHAD: 8个32位通用寄存器依次进栈

⑦ **POPA/POPAD** 所有 (16/32位 P63) 寄存器出栈指令

➤ **PUSHA、POPA** 80286以上机器

➤ **PUSHAD、POPAD** 80386以上

⑧ XCHG 交换指令

格式: XCHG OPR1, OPR2

执行的操作 (OPR1) \leftrightarrow (OPR2)

- 操作数可以是8位、16位、32位
- 该指令可能的组合是:

XCHG 寄存器操作数, 寄存器操作数

XCHG 寄存器操作数, 存储器操作数

XCHG 存储器操作数, 寄存器操作数

XCHG AL, BH

XCHG BX, [BP+SI]

XCHG [BP+SI], BX

注意:

- * 不影响标志位
- * 不允许使用段寄存器

3.3.1.2 累加器专用传送指令

- | | | |
|--------|----------|-------|
| 1. IN | (input) | I/O输入 |
| 2. OUT | (output) | I/O输出 |

只限于使用EAX、AX或AL

3. XLAT (translate)

只限于使用AL和BX

1、 I/O输入指令

- ◆ **格式：** IN ACR, PORT
- ◆ **功能：**把外设端口（PORT）的内容传送给累加器（ACR）
 - 可以传送8位、16位、32位, 相应的累加器选择AL、AX、EAX
- ◆ 端口号在0~255之间, 则端口号直接写在指令中
 - 长格式: IN AL, PORT (字节)
IN AX, PORT (字)
 - 执行操作: (AL) ← (PORT) (字节)
(AX) ← (PORT+1, PORT) (字)
- ◆ 端口号大于255, 则端口号通过DX寄存器间接寻址, 即端口号应先放入DX中
 - 短格式: IN AL, DX (字节)
IN AX, DX (字)
 - 执行操作: (AL) ← (DX) (字节)
(AX) ← (DX+1, DX) (字)

2. I/O输出指令 OUT

- ◆ **格式:** OUT PORT, ACR
- ◆ **功能:** 把累加器的内容传送给外设端口, 对累加器和端口号的选择限制同IN指令

长格式: OUT PORT, AL (字节)
 OUT PORT, AX (字)
执行操作: (PORT) ← (AL) (字节)
 (PORT+1, PORT) ← (AX) (字)

短格式: OUT DX, AL (字节)
 OUT DX, AX (字)
执行操作: ((DX)) ← (AL) (字节)
 ((DX+1, DX)) ← AX (字)

注意：

- * 不影响标志位
- * 前256个端口号00H~FFH可直接在指令中指定(长格式)
- * 如果端口号 ≥ 256 ，端口号 \rightarrow DX (短格式)

例： IN AX, 28H ; MOV DX, 28H
 ; IN AX, DX
 MOV DATA_WORD, AX

例： MOV DX, 3FCH 例： OUT 5, AL
 IN AX, DX

例： 测试某状态寄存器（端口号27H）的第2位是否为1

```
IN      AL, 27H
TEST    AL, 00000100B
JNZ     ERROR                                ;若第2位为1，转到ERROR处理
```

3. 换码指令（查表转换指令）

- ◆ 换码指令：XLAT 或 XLAT OPR
- ◆ 执行操作 $(AL) \leftarrow ((BX) + (AL))$
 - OPR一般是表格的起始符号基址，只是为了程序可读性，执行时只能使用BX、AL
 - 这条指令前，一定要使BX指向表格的起始符号基址
 - 只限于使用AL和BX
- ◆ 换码指令常用于把一种代码转换为另一种代码

注意：

- * 不影响标志位
- * 字节表格（长度不超过256）首地址的偏移地址 → BX
- * 需转换代码 → AL

字符	ASCII码
0~9	30H~39H
A~Z	41H~5AH
a~z	61H~7AH

例：十进制3转换成它的ASCII码

假设DS=F000H

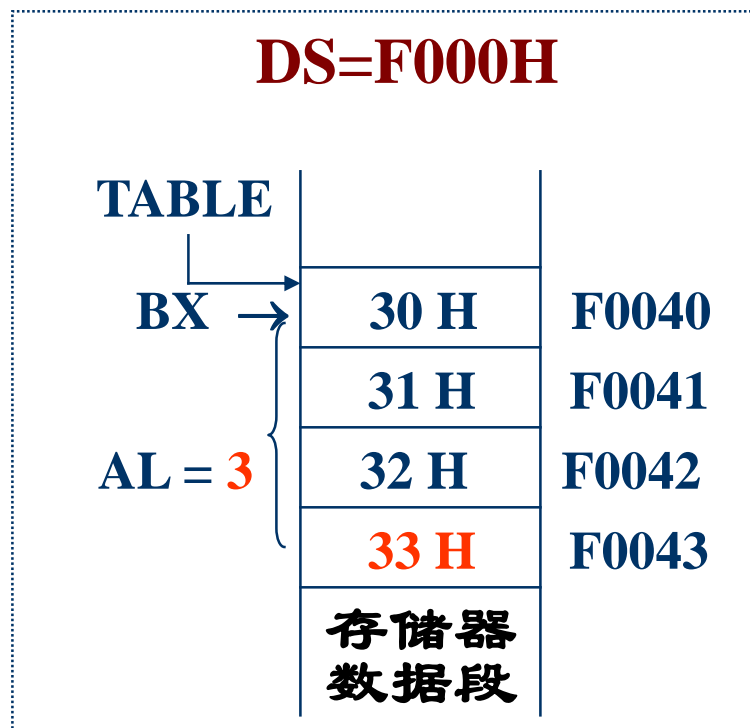
MOV BX, OFFSET TABLE ; BX=0040H

MOV AL, 3

XLAT TABLE ; (AL) ← ((BX) + (AL))

• $DS*16 + BX + AL = F0043H$

• 指令执行后 **AL=33H**



3.3.1.3 地址传送指令

- | | |
|--------------------------------|-----------|
| 1. LEA(load effective address) | 有效地址送寄存器 |
| 2. LDS(load DS with pointer) | 指针送寄存器和DS |
| 3. LES(load ES with pointer) | 指针送寄存器和ES |
| 4. LSS(load SS with pointer) | 指针送寄存器和SS |
| 5. LFS(load FS with pointer) | 指针送寄存器和FS |
| 6. LGS(load GS with pointer) | 指针送寄存器和GS |

- 这组指令完成把操作数地址送到指定的寄存器
- LEA是把源操作数的有效地址EA送到指定的寄存器中
- 其余5条指令源操作数只能用存储器寻址方式，对于16位指针寄存器而言，低地址中的16位数据装入指针寄存器，高地址中的16位数据装入段寄存器

● 地址传送指令

有效地址送寄存器指令: **LEA REG, SRC**
执行操作: **(REG) ← SRC**

把源操作数有效地址送到指定的寄存器中

指针送寄存器和DS指令: **LDS REG, SRC**
执行操作: **(REG) ← (SRC)**
 (DS) ← (SRC+2)

相继二字 → 寄存器、DS

指针送寄存器和ES指令: **LES REG, SRC**
执行操作: **(REG) ← (SRC)**
 (ES) ← (SRC+2)

相继二字 → 寄存器、ES

例1: LEA BX, [BX+SI+0F62H] ; 操作数的有效地址BX+SI+0F62H送BX

例2: LDS SI, [10H] ;操作数低16位送SI, 高16位送DS

例3: LES DI, [BX] ;操作数低16位送DI, 高16位送ES

例4:

TABLE DS:1000H	存储器 数据段	低 16 位
	40 H	
	00 H	高 16 位
	00 H	
	30 H	

MOV BX, TABLE ; BX=0040H

MOV BX, OFFSET TABLE ; BX=1000H

LEA BX, TABLE ; BX=1000H

LDS BX, TABLE ; BX=0040H
; DS=3000H

LES BX, TABLE ; BX=0040H
; ES=3000H

LSS BX, TABLE ; BX=0040H
; SS=3000H

注意:

- * 不影响标志位
- * REG不能是段寄存器
- * SRC必须为存储器寻址方式

MOV指令就可以实现这些功能

为什么使用地址传送指令

- ◆ **简单：** 多个数据块操作时，指针改变非常方便
- ◆ **安全：** 可以在一个不被中断的指令操作中同时改变标识地址的两个寄存器，确保在一条指令中使段寄存器和指针寄存器都被重置

例如 SS、SP改变时特别重要

3.3.1.4 标志寄存器传送指令

LAHF (load AH with flags)

标志送AH

SAHF (store AH into flags)

AH送标志寄存器

PUSHF/PUSHFD (push the flags or eflags)

标志进栈

POPF/POPFD (pop the flags or eflags)

标志出栈

其中指令中的POPF、POPFD、SAHF指令影响标志位，其他不影响

.....		ID	VIP	VIF	AC	RF	VM		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	
31...	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

标志送AH指令： LAHF

执行操作： (AH) \leftarrow (PSW的低字节)

AH送标志寄存器指令： SAHF

执行操作： (PSW的低字节) \leftarrow (AH)

标志进栈指令： PUSHF

执行操作： (SP) \leftarrow (SP) - 2
((SP+1, SP)) \leftarrow (PSW)

标志出栈指令： POPF

执行操作： (PSW) \leftarrow ((SP+1, SP))
(SP) \leftarrow (SP) + 2

PUSHFD、POPF D对应32位EFLAGS

3.3.1.5 类型转换指令

1. CBW(convert byte to word)

字节转换为字，AL中的内容符号扩展到AH

2. CWD/CWDE

字转换为双字

- CWD: AX的内容符号扩展到DX, 形成DX:AX中的双字
- CWDE: AX的内容符号扩展到EAX, 形成EAX中的双字

3. CDQ

- 双字转换为4字指令
- EAX的内容符号扩展到EDX, 形成EDX:EAX中的4字

4. BSWAP (486及后继机型使用)

- 字节交换指令, 字节次序变反
- 例如 EAX=11 22 33 44H
 - 执行 BSWAP EAX 后 EAX=44 33 22 11H

这些指令只对累加器内容进行带符号扩展和处理

3.3.2 算术指令

这类指令包括二进制运算指令和十进制运算指令：

1. **加法指令**（加法指令ADD、带进位加法指令ADC、加1指令INC等）
2. **减法指令**（减法指令SUB、带借位减法指令SBB、减1指令DEC、求补指令NEG、比较指令CMP等）
3. **乘法指令**（无符号乘法指令MUL、带符号数乘法指令IMUL等）
4. **除法指令**（无符号除法指令DIV、带符号除法指令IDIV）
5. **十进制调整指令**（DAA、DAS等）

注意：

- 目标操作数不允许是立即数和CS段寄存器
- 两个操作数不能同时为存储器操作数
- 单操作数指令不允许使用立即数方式
- 除十进制调整指令外，其他指令均影响某些标志位

3.3.2.1 加法指令

- ① **ADD** (add) **加法指令**
- ② **ADC** (add with carry) **带进位加法指令**
- ③ **INC** (increment) **加1指令**

- ④ **XADD** (exchange and add) **交换并相加指令**
 - 该指令只能用于486及后继机型, 它把目的操作数装入源, 并把源和目的操作数之和送目的地址

看 p 70~71 加法运算后对OF和CF的影响情况

(1) 加法指令 ADD

格式: ADD DST, SRC

功能: $(DST) + (SRC) \rightarrow (DST)$

说明: 对操作数的限定同MOV指令

(2) 带进位加法指令 ADC

格式: ADC DST, SRC

功能: $(DST) + (SRC) + CF \rightarrow (DST)$

说明: 对操作数的限定同MOV指令, 该指令适用于多字节或多字的加法运算

(3) 加1指令 INC

格式: INC OPR

功能: $(\text{OPR}) + 1 \rightarrow (\text{OPR})$

说明: 很方便地实现地址指针或循环次数的加1修改

(4) 互换并加法指令 XADD (486以上)

格式: XADD DST, SRC

功能: $(\text{SRC}) + (\text{DST}) \rightarrow \text{暂存器}$

$(\text{DST}) \rightarrow (\text{SRC})$

暂存器 $\rightarrow (\text{DST})$

说明: 该指令执行后, 原DST的内容在SRC中, 和在DST中

加法指令对条件标志位的影响

- * 条件标志位最主要的4位：CF, ZF, SF, OF
- * 除INC指令不影响CF标志外，均对条件标志位有影响

SF= $\begin{cases} 1 & \text{结果为负} \\ 0 & \text{否则} \end{cases}$

ZF= $\begin{cases} 1 & \text{结果为0} \\ 0 & \text{否则} \end{cases}$

CF= $\begin{cases} 1 & \text{和的最高有效位 有 向高位的进位} \\ 0 & \text{否则} \end{cases}$

OF= $\begin{cases} 1 & \text{两个操作数符号相同，而结果符号与之相反} \\ 0 & \text{否则} \end{cases}$

CF 位表示 无符号数 相加的溢出

OF 位表示 带符号数 相加的溢出

举例: $n=8$ bit 带符号数 ($-128 \sim 127$), 无符号数 ($0 \sim 255$)

$$\begin{array}{r} 0000\ 0100 \\ +\ 0000\ 1011 \\ \hline 0000\ 1111 \end{array}$$

带: $(+4) + (+11) = +15$ OF=0

无: $4 + 11 = 15$ CF=0

带符号数和无符号数都不溢出

$$\begin{array}{r} 1000\ 0111 \\ +\ 1111\ 0101 \\ \hline 1\ 0111\ 1100 \end{array}$$

带: $(-121) + (-11) = +124$ OF=1

无: $135 + 245 = 124$ CF=1

带符号数和无符号数都溢出

$$\begin{array}{r} 0000\ 0111 \\ +\ 1111\ 1011 \\ \hline 1\ 0000\ 0010 \end{array}$$

带: $(+7) + (-5) = +2$ OF=0

无: $7 + 251 = 2$ CF=1

无符号数溢出

$$\begin{array}{r} 0000\ 1001 \\ +\ 0111\ 1100 \\ \hline 1000\ 0101 \end{array}$$

带: $(+9) + (+124) = -123$ OF=1

无: $9 + 124 = 133$ CF=0

带符号数溢出

例：双精度数的加法

(DX) = 0002H (AX) = 0F365H

(BX) = 0005H (CX) = 8100H

指令序列 **ADD AX, CX** ; (1)

ADC DX, BX ; (2)

(1) 执行后, **(AX) = 7465H**

CF=1 OF=1 SF=0 ZF=0

(2) 执行后, **(DX) = 0008H**

CF=0 OF=0 SF=0 ZF=0

3.3.2.1 减法指令

- | | | |
|---|------------------|-------------------------|
| ① | SUB | 减法 |
| ② | SBB | 带借位减法 |
| ③ | DEC | 减1 |
| ④ | NEG | 求补 |
| ⑤ | CMP | 比较 |
| ⑥ | <i>CMPXCHG</i> | 比较并交换 (486以后) |
| ⑦ | <i>CMPXCHG8B</i> | 比较并交换 8 字节 (Pentium 以后) |

(1) 减法指令 SUB

格式: SUB DST, SRC

功能: $(DST) - (SRC) \rightarrow (DST)$

说明: 除是实现减法功能外, 其他要求同ADD

(2) 带借位减法指令 SBB

格式: SBB DST, SRC

功能: $(DST) - (SRC) - CF \rightarrow (DST)$

说明: 除了操作为减外, 其他要求同ADC, 该指令适用于多字节或多字的减法运算

(3) 减1指令 DEC

格式: DEC OPR

功能: $(OPR) - 1 \rightarrow (OPR)$

说明: 可以很方便地实现地址指针或循环次数的减1修改

(4) 求补指令 NEG

格式: NEG OPR

功能: $0FFFFH - (OPR) + 1 \rightarrow (OPR)$

对目标操作数（含符号位）求反加1，并且把结果送回目标

说明: 利用NEG指令可实现求一个数的补码

(5) 比较指令 CMP

格式: CMP OPR1, OPR2

功能: $(OPR1) - (OPR2)$ ，只影响标志位，不影响源和目的操作数

说明: 这条指令执行相减操作后只根据结果设置标志位，并不改变两个操作数的原值，其他要求同SUB。CMP指令常用于比较两个数的大小。

减法指令对条件标志位（CF/OF/ZF/SF）的影响

* 除DEC指令不影响CF标志外，均对条件标志位有影响。

CF = $\begin{cases} 1 & \text{被减数的最高有效位 有 向高位的借位} \\ 0 & \text{否则} \end{cases}$
或

CF = $\begin{cases} 1 & \text{减法转换为加法运算时 无 进位} \\ 0 & \text{否则} \end{cases}$

OF = $\begin{cases} 1 & \text{两个操作数符号相反，而结果的符号与减数相同} \\ 0 & \text{否则} \end{cases}$

CF 位表示 无符号数 减法的溢出。

OF 位表示 带符号数 减法的溢出。

例 3.54, 3.55
(P73)

NEG 指令对 CF/OF 的影响

$$CF = \begin{cases} 0 & \text{操作数为0} \\ 1 & \text{否则} \end{cases}$$

$$OF = \begin{cases} 1 & \text{操作数为 -128 (字节运算) 或} \\ & \text{操作数为 -32768 (字运算)} \\ 0 & \text{否则} \end{cases}$$

(6) 比较并交换指令 CMPXCHG

80486及其后续机型

格式: CMPXCHG DST, SRC

SRC: AL、AH、AX寄存器; DST: 寄存器或存储器寻址方式

功能: (DST) \leftarrow (SRC), 影响标志位; 如果相等, (SRC) \rightarrow (DST); 不相等, (DST) \rightarrow (SRC)

说明: 这条指令执行相减操作后根据结果设置标志位

(7) 比较并交换 8 字节指令 CMPXCHG8B

Pentium及其后续机型

格式: CMPXCHG8B DST, SRC

SRC: EDX:EAX构成的64位; DST: 存储器寻址方式确定的64位

功能: (DST) \leftarrow (SRC), 影响标志位; 如果相等, (SRC) \rightarrow (DST); 不相等, (DST) \rightarrow (SRC)

例：x、y、z 均为双精度数，分别存放在地址为X, X+2; Y, Y+2; Z, Z+2的存储单元中，用指令序列实现 $w \leftarrow x + y + 24 - z$ ，并用W, W+2单元存放w (P74)

```
MOV  AX,  X
MOV  DX,  X+2
ADD  AX,  Y
ADC  DX,  Y+2      ;  x+y
ADD  AX,  24
ADC  DX,  0        ;  x+y+24
SUB  AX,  Z
SBB  DX,  Z+2      ;  x+y+24-z
MOV  W,  AX
MOV  W+2, DX      ;  结果存入W, W+2单元
```

3.3.2.3 乘法指令

① MUL 无符号数乘法

② IMUL 带符号数乘法

- 对除CF和OF以外的条件码无定义，即不确定
- 在乘法指令里，目的操作数必须是累加器（字运算为AX，字节运算为AL）
- 两个8位数相乘得到的是16位乘积，两个16位数相乘得到的是32位乘积，存放在DX:AX中

(1) 无符号数乘法 MUL

格式: MUL SRC ; SRC: 除立即数以外的寻址方式

功能: 字节操作: $(AL) * (SRC) \rightarrow (AX)$

字操作: $(AX) * (SRC) \rightarrow (DX:AX)$

双字操作: $(EAX) * (SRC) \rightarrow (EDX:EAX)$

※ 乘积的高一半为0, 则CF、OF均为0, 否则CF、OF均为1
这样可以检查结果是字节、字或双字

(2) 带符号数乘法 IMUL

格式: IMUL SRC ; SRC: 除立即数以外的寻址方式

功能: 字节操作: $(AL) * (SRC) \rightarrow (AX)$

字操作: $(AX) * (SRC) \rightarrow (DX:AX)$

双字操作: $(EAX) * (SRC) \rightarrow (EDX:EAX)$

※ 乘积的高一半是低一半的符号扩展, 则CF、OF均为0, 否则CF、OF均为1。其实质和MUL情况下一样, 主要用于判断结果是字节、字或双字

乘法指令对 CF/OF 的影响

MUL指令: CF,OF = $\begin{cases} 00 & \text{乘积的高一半为零} \\ 11 & \text{否则} \end{cases}$

IMUL指令: CF,OF = $\begin{cases} 00 & \text{乘积的高一半是低一半的符号扩展} \\ 11 & \text{否则} \end{cases}$

(3) 多操作数带符号数乘法 IMUL

80286及其后续机型，增加了双操作数和三操作数指令

- 格式：IMUL REG, SRC

SRC: 任一种寻址方式，立即数可以8位（自动符号扩展）、16位

功能： 字操作： $(REG16) * (SRC) \rightarrow (REG16)$

双字操作： $(REG32) * (SRC) \rightarrow (REG32)$

- 格式：IMUL REG, SRC, IMM

SRC: 除立即数外的寻址方式

IMM: 立即数， 8位（自动符号扩展）、16位、32位，**与目的操作数长度一致**

功能： 字操作： $IMM * (SRC) \rightarrow (REG16)$

双字操作： $IMM * (SRC) \rightarrow (REG32)$

3.3.2.4 除法指令

① DIV 无符号数除法

② IDIV 带符号数除法

- 对所有条件码无定义，即不确定
- 源操作数可以用除立即数以外的任何一种寻址方式

(1) 无符号除法指令 DIV

格式: DIV SRC (DIV reg DIV mem)

- 源操作数不能是立即数；被除数必须事前放在隐含的寄存器中；可以实现8位、16位、32位无符号数除
- 具体操作为：

字节型除法: $(AX) \div (SRC)_8 \rightarrow \begin{cases} \text{商: (AL)} \\ \text{余数: (AH)} \end{cases}$

字型除法: $(DX:AX) \div (SRC)_{16} \rightarrow \begin{cases} \text{商: (AX)} \\ \text{余数: (DX)} \end{cases}$

双字型除法: $(EDX:EAX) \div (SRC)_{32} \rightarrow \begin{cases} \text{商: (EAX)} \\ \text{余数: (EDX)} \end{cases}$

(2) 带符号除法指令 IDIV

格式: IDIV SRC (IDIV reg IDIV mem)

- 实现两个带符号数相除，商和余数均也为带符号数，余数符号与被除数相同
- 源操作数不能是立即数；被除数必须事前放在隐含的寄存器中；可以实现8位、16位、32位无符号数除
- 具体操作为：

字节型除法: $(AX) \div (SRC)_8 \rightarrow \begin{cases} \text{商: (AL)} \\ \text{余数: (AH)} \end{cases}$

字型除法: $(DX:AX) \div (SRC)_{16} \rightarrow \begin{cases} \text{商: (AX)} \\ \text{余数: (DX)} \end{cases}$

双字型除法: $(EDX:EAX) \div (SRC)_{32} \rightarrow \begin{cases} \text{商: (EAX)} \\ \text{余数: (EDX)} \end{cases}$

注意：

(1) 除法指令：除数、商和余数的位数是被除数的一半

- 字节操作时，要求被除数为16位，商为8位
- 字操作时，要求被除数为32位，商为16位
- 双字操作时，要求被除数为64位，商为32位

(2) 若除法结果产生溢出（如：字节操作被除数的高8位绝对值 \geq 除数的绝对值，这时商会超过8位，则商就会产生溢出），80X86是由系统直接转入0号中断处理

例：x, y, z, v 均为16位带符号数，计算
 $(v - (x * y + z - 540)) / x$ (P78)

```
MOV    AX, X
IMUL   Y           ;  x*y → (DX, AX)
MOV    CX, AX
MOV    BX, DX      ;  结果暂存 (BX, CX)
MOV    AX, Z
CWD                    ;  Z → (DX, AX)
ADD    CX, AX
ADC    BX, DX       ;  x*y+z → (BX, CX)
SUB    CX, 540
SBB    BX, 0        ;  x*y+z-540
MOV    AX, V
CWD                    ;  V → (DX, AX)
SUB    AX, CX
SBB    DX, BX       ;  v-(x*y+z-540)
IDIV   X            ;  (v-(x*y+z-540))/x → (AX)
                        余数 → (DX)
```

3.3.2.5 十进制调整指令

为便于十进制计算，80X86还提供了一组十进制调整指令，这组指令在二进制计算的基础上，给予十进制调整，可以直接得到十进制的结果

(1) 压缩的BCD码调整指令

DAA DAS

(2) 非压缩的 BCD码调整指令

AAA AAS AAM AAD

◆ AAA(ASCII adjust after addition)

● BCD码

BCD码：用二进制编码的十进制数，又称二—十进制数

压缩的BCD码：用 4 位二进制数表示 1 位十进制数

$$\text{例：} (59)_{10} = (0101\ 1001)_{\text{BCD}}$$

非压缩的BCD码：用 8 位二进制数表示 1 位十进制数

$$\text{例：} (59)_{10} = (uuuu\ 0101\ uuuu\ 1001)_{\text{BCD}}$$

数字的 ASCII 码是一种 非压缩的 BCD 码

DIGIT	ASCII	BCD
0	30H	0011 0000
1	31H	0011 0001
2	32H	0011 0010
...
9	39H	0011 1001

例：写出 $(3590)_{10}$ 的压缩 BCD 码和非压缩 BCD 码，并分别把它们存入数据区 **PAKED** 和 **UNPAK**

压缩BCD: $(3590)_{10} = (0011\ 0101\ 1001\ 0000)_{BCD}$

非压缩BCD:

$(3590)_{10} = (00000011\ 00000101\ 00001001\ 00000000)_{BCD}$

PAKED	
	90H
	35H

UNPAK	
	00H
	09H
	05H
	03H

十进制调整指令

问题的提出:

$$\begin{array}{r} 19 \quad \text{压缩BCD:} \quad 0001 \ 1001 \\ + 08 \\ \hline 27 \end{array}$$

$$\begin{array}{r} 0001 \ 1001 \\ + 0000 \ 1000 \\ \hline 0010 \ 0001 + 110 \end{array}$$

$(0010 \ 0111)_{\text{BCD}}$ ← $\text{AF}=1$

P79

注意：

- ◆ 由于仅仅是调整而不是真正意义上的十进制运算，所以这组指令都需要与相应的二进制运算指令配合才可得到正确的十进制结果
- ◆ 使用这组指令时，要注意参与运算的操作数必须是十进制数的BCD编码形式

(1) 压缩的BCD码调整指令

- ◆ DAA 加法的压缩BCD码调整指令

格式：DAA

功能：AL中的和调整为压缩BCD码 → AL

➤ 必须跟在两个压缩BCD码的二进制加法指令ADD/ADC之后

- 结果：AL中有2位压缩的BCD码
- 0F位无定义，但影响其他标志位

调整规则：

- 如和的数值在1010-1111之间或者AF标志为1，则加6 (0110) 就可得到正确结果

调整指令操作（按调整规则系统自动执行）：

第一步：调整低位

```
IF ((AL AND 0FH) > 09H) OR (AF = 1)
THEN
    AL ← AL + 06H;
    /*十进制个位数加6调整成压缩BCD码*/
    AF ← 1;
```

第二步：调整高位

```
IF ((AL AND 0F0H) > 90H) OR (CF = 1)
THEN
    AL ← AL + 60H;
    /*十进制十位数加6调整成压缩BCD码*/
    CF ← 1;
```

例3.64 (p80)

◆ DAS 减法的压缩BCD码调整指令

格式:DAS

功能: AL中的差调整为压缩BCD码 → AL

➤ 必须跟在两个压缩BCD码的二进制**减法指令SUB/SBB之后**

- 结果: AL中有2位压缩的BCD码
- 0F位无定义, 但影响其他标志位

调整指令操作（按调整规则系统自动执行）：

第一步：调整低位

IF ((AL AND 0FH) > 09H) OR (AF = 1)

THEN

AL ← AL - 06H;

/*十进制个位数减6调整成压缩BCD码*/

AF ← 1;

第二步：调整高位

IF ((AL AND 0F0H) > 90H) OR (CF = 1)

THEN

AL ← AL - 60H;

/*十进制十位数减6调整成压缩BCD码*/

CF ← 1;

为什么减6？ 因为十进制借位应该当10用，而SUB把借位当16用少减了6

(2) 非压缩的BCD码调整指令

- ◆ **AAA 加法的ASCII码调整指令**
 - AL中的和调整为非压缩BCD码 → AL
 - AH + 调整产生的进位值 → AH
 - 影响AF、CF，其他标志位无定义
- ◆ **AAS 减法的ASCII码调整指令**
 - AL中的差调整为非压缩BCD码 → AL
 - AH - 调整产生的借位值 → AH
 - 影响AF、CF，其他标志位无定义

◆ AAM 乘法的ASCII码调整指令

- AL中的积调整为非压缩BCD码 → AX
- AL/OAH, 商→ AH, 余数→ AL
- AF、CF和OF位无定义, 影响SF、ZF和PF

◆ AAD 除法的ASCII码调整指令

- 用在除法指令前使用, 为除法指令作好准备
- $10 * (AH) + AL$ (变成二进制数) → AL, 0 → AH
- AF、CF和OF位无定义, 影响SF、ZF和PF

压缩BCD运算举例：

(1) **MOV AL, BCD1** ; BCD1=34H
ADD AL, BCD2 ; BCD2=59H, (AL)=8DH
DAA ; 8DH+06H=93H
MOV BCD3, AL ; BCD3=93H

(2) **MOV AL, BCD1** ; BCD1=34H
SUB AL, BCD2 ; BCD2=59H, (AL)=0DBH
DAS ; 0DBH-60H-06H=75H
MOV BCD3, AL ; BCD3= 75 = - 25 (10ⁿ补码)

非压缩BCD运算举例:

(1) **MUL BL ; (AX)=(AL) × (BL)=08 × 09**
AAM ; (AL)/0AH= 48H /0AH → 0702

(2) **AAD ; (AX) → (AH) × 0AH + (AL)=48H**
DIV BL ; (AL) = (AX)/(BL)=48H/4=12H
AAM ; (AL)/0AH=12H/0AH=0108

3.3.3 逻辑指令

逻辑指令包括：

1. 逻辑运算指令
2. 位测试并修改指令
3. 位扫描指令
4. 移位指令

1. 逻辑运算指令

逻辑运算指令可以对字或字节执行逻辑运算，386及其后继机型还可以执行双字操作

基本指令如下：

AND (and)	逻辑与
OR (or)	逻辑或
NOT (not)	逻辑非
XOR (exclusive or)	异或
TEST (test)	测试

名称	格 式	功 能	标 志
逻辑非	NOT DST	(DST) 按位变反 送DST	不影响
逻辑与	AND DST, SRC	$(DST) \wedge (SRC)$ $\rightarrow (DST)$	CF和OF清0, 影响SF、ZF及PF, AF无定义
测 试	TEST OPR1,OPR2	$(OPR1) \wedge (OPR2)$	同AND指令
逻辑或	OR DST, SRC	$(DST) \vee (SRC)$ $\rightarrow (DST)$	同AND指令
逻辑 异或	XOR DST, SRC	$(DST) \vee (SRC)$ $\rightarrow (DST)$	同AND指令

- ◆ **NOT指令不允许立即数**
- ◆ 其他指令操作数寻址方式与MOV指令的限制相同
 - 一个操作数放在寄存器中，一个数据用任意寻址方式得到
- ◆ 逻辑与和逻辑测试的区别在于后者执行后只影响标志位而不改变任何操作数本身

逻辑运算指令用途

- ◆ **逻辑非指令**：可用于把操作数的每一位变反
- ◆ **逻辑与指令**：用于把某位清0（和0相与，也可称为屏蔽某位）、某位保持不变（和1相与）
- ◆ **逻辑或指令**：用于把某位置1（与1相或）、某位保持不变（与0相或）
- ◆ **逻辑异或指令**：用于把某位变反（与1相异或）、某位保持不变（与0相异或）
- ◆ **逻辑测试指令**：可用于只测试某位值而不改变操作数

MOV AL, 00001111B ;AL=00001111B

NOT AL ;AL=11110000B

AND AL, 0FH ;清0高4位, 低4位不变

OR AL, 30H ;D5、D4位置1(与1相或)、其他位不变

IN AL, 61H

XOR AL, 2 ; 00000010

OUT 61H, AL ; 使61H端口的D1位变反

XOR AX, AX ; AX寄存器清0

例：屏蔽AL的第0、1两位

AND AL, 0FCH

```
      * * * * *
AND  1 1 1 1 1 1 0 0
      * * * * * 0 0
```

例：置AL的第5位为1

OR AL, 20H

```
      * * * * *
OR   0 0 1 0 0 0 0 0
      * * 1 * * * * *
```

例：使AL的第0、1位变反

XOR AL, 3

```
      * * * * * 0 1
XOR  0 0 0 0 0 0 1 1
      * * * * * 1 0
```

例：测试某些位是0是1

TEST AL, 1
JZ EVEN

```
      * * * * *
AND  0 0 0 0 0 0 0 1
      * * * * *
      0 0 0 0 0 0 0 *
```

2. 位测试并修改指令

- ◆ 386及其后继机型增加了本组指令

BT	位测试
BTS	位测试并置 1
BTR	位测试并置 0
BTC	位测试并变反

名称	格 式	功 能	标 志
位测试	BT DST, SRC	测试由SRC指定的 DST中的位	所选位值送CF, 其他标志无定义
位测试 并置位	BTS DST, SRC	测试并置1由SRC 指定的DST中的位	同上
位测试 并复位	BTR DST, SRC	测试并清0由SRC 指定的DST中的位	同上
位测试 并取反	BTC DST, SRC	测试并取反由SRC 指定的DST中的位	同上

- ◆ 首先把指定位的值送给CF标志，然后对该位按照指令的要求操作
- ◆ 目标可以是除立即数以外的任一种寻址方式
- ◆ 目标操作数的位置从最右边位开始、从0开始计数
- ◆ 源可以是8位的**立即数**、**寄存器**
 - 若源操作数是立即数，则其值不应超过目标操作数的长度
 - 也可用任一字寄存器或双字寄存器给出同一个值

MOV AX, 2357H

;AX=0010 0011 0101 0111B

BT AX, 0 ;CF=1, AX=2357H

;AX=0010 0011 0101 0111B

;AX=0010 0011 0101 0111B

BTC AX, 3 ;CF=0, AX= 235FH

;AX=0010 0011 0101 1111B

3. 位扫描指令

- ◆ 386及其后继机型增加了本组指令

BSF(bit scan forward)

正向位扫描（从最低位开始）

BSR(bit scan reverse)

反向位扫描（从最高位开始）

**记录正向/反向检索到的第一个“1”的位置，
记录在目标寄存器中**

0010101001011000

(1) 正向位扫描指令 BSF

格式: BSF REG, SRC

功能: 从位0开始, **从右向左**扫描SRC操作数中第一个是1的位

- ◆ 若遇到第一个为1的位, 则ZF置0, 并把该位的位号送REG
- ◆ 若SRC=0, 则ZF置1, REG值不确定
- REG只能是字或双字通用寄存器; SRC不能是立即数, 其他寻址方式均可
- 其他标志位无定义

ZF=0, 扫描到1, 位置记录在REG中
ZF=1, 没有扫描到1, 即SRC=0

(2) 反向位扫描指令 BSR

格式: BSR REG, RSC

功能: 该指令从最高位开始, **从左向右**扫描, 其他同BSF

4. 移位指令

(1) 移位指令（基本移位指令）

SHL SAL SHR SAR

(2) 循环移位指令





ROL ROR RCL RCR

(3) 双精度移位指令（386及后继机型使用本组指令）

SHLD 双精度左移指令

SHRD 双精度右移指令

(1) 移位指令

名 称	格 式	功 能	标 志
逻辑左移	SHL DST, CNT		CF中总是最后移出的一位； ZF、SF、PF按结果设置；当CNT= 1时, 移位使DST最高位变化0F置1, 否则清0
算术左移	SAL DST, CNT		同上
逻辑右移	SHR DST, CNT		同上
算术右移	SAR DST, CNT		同上

说明： 逻辑移位看作无符号数移位， 算术移位看作带符号数移位

DST可以是8位、16位或32位的寄存器或存储器操作数

CNT是移位位数计数器

8086 / 8088对CNT的限定：



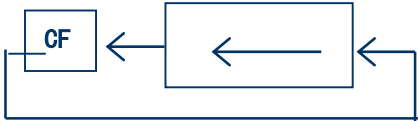
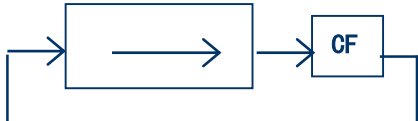
当CNT=1时，直接写在指令中； 当CNT>1时，由CL寄存器给出

其他机型对CNT的限定：

当CNT=1，或8位立即数（1-31）时，直接写在指令中； 也可由CL寄存器给出

- ◆ SHL和SAL指令的功能相同，在机器中实际上它们对应的是同一种操作
- ◆ 使用这组指令
 - 可以实现基本的移位操作
 - 可以用于对一个数进行 2^n 的倍增或倍减运算，比直接使用乘除法效率要高得多

(2) 循环移位指令

名 称	格 式	功 能	标 志
循环左移	ROL DST, CNT		不影响CF、OF以外的其它条件标志；CF中总是最后移进的位；当CNT=1时，移位使DST符号位改变则OF置1，否则清0
循环右移	ROR DST, CNT		同上
带进位循环左移	RCL DST, CNT		同上
带进位循环右移	RCR DST, CNT		同上

对DST和CNT的限定同基本移位指令

注意:

- * **OPR**可用除立即数以外的任何寻址方式

- * **CNT=1, SHL OPR, 1**

- CNT>1, MOV CL, CNT**

- SHL OPR, CL ; 以SHL为例**

- * **条件标志位:**

- CF = 移入的数值**

- $$OF = \begin{cases} 1 & \text{CNT=1时, 最高有效位的值发生变化} \\ 0 & \text{CNT=1时, 最高有效位的值不变} \end{cases}$$

- 移位指令:**

- SF、ZF、PF 根据移位结果设置, AF无定义**

- 循环移位指令:**

- 不影响 SF、ZF、PF、AF**

例：(AX)= 0012H, (BX)= 0034H, 把它们装配成(AX)= 1234H

```
MOV    CL, 8
ROL    AX, CL
ADD    AX, BX
```

例：(BX) = 84F0H

(1) (BX) 为无符号数, 求 (BX) / 2

```
SHR    BX, 1          ; (BX) = 4278H
```

(2) (BX) 为带符号数, 求 (BX) × 2

```
SAL    BX, 1          ; (BX) = 09E0H, OF=1
```

(3) (BX) 为带符号数, 求 (BX) / 4

```
MOV    CL, 2
SAR    BX, CL          ; (BX) = 0E13CH
```

(3) (BX)=84F0H, 把 (BX) 中的 16 位数每 4 位压入堆栈

```
MOV    CH, 4           ; 循环次数
MOV    CL, 4           ; 移位次数
```

NEXT:


```
ROL    BX, CL
MOV    AX, BX
AND    AX, 000FH
PUSH   AX
DEC    CH
JNZ    NEXT
```

0000	← (SP)
000F	
0004	
0008	

(3) 双精度移位指令

(1) 双精度左移指令 SHLD (386及其后继机型)₀

格式: SHLD DST, REG, CNT



功能: 把操作数DST左移由CNT指定的位(设为n), 空出的位用REG高
端的n位填充, REG的内容不变, 最后移出的位在进位标志CF中

- DST: 16或32位的寄存器或存储器操作数
- REG: 与DST长度相同的寄存器
- CNT是移位位数: 8位立即数, 或者CL内容
 - 8位立即数, 提供0-31之间的值, 大于31时机器自动 MOD32
 - CL内容指定
- 结果标志位:
 - CF=最后移出的一位
 - 如果CNT=1, 当移位后DST最高位发生变化时, OF置1, 否则清0;
当CNT>1时OF值无定义
 - SF、ZF、PF按照DST结果设置
 - AF除移位次数为0外无定义

(2) 双精度右移指令 SHRD (386及其后继机型)

格式: SHRD DST, REG, CNT

功能: 把操作数DST右移由CNT指定的位, 空出的位用REG低端的n位填充, REG的内容不变, 最后移出的位在进位标志CF中

- 其他同SHLD



MOV BX, 8321H ; 1000 0011 0010 0001

MOV CX, 5678H ; 0101 0110 0111 1000

SHLD BX, CX, 1 ; 0000 0110 0100 0010

; BX=0642H, CX=5678H, CF=1, OF=1



SHRD BX, CX, 2 ; 0010 0000 1100 1000

; BX=20B8H, CX=5678H, CF=0, OF=U



3.3.4 串处理指令

- ◆ 处理存放在存储器里的**数据串**，所有串指令都可以处理**字节或字**，386及后继机型还可以处理**双字**
- ◆ 利用串操作指令可以直接处理两个存储器单元的操作数，方便地处理**字符串或数据块**
- ◆ 串处理指令包括：

MOVS	串传送	CMPS	串比较
SCAS	串扫描	LODS	从串取
STOS	存入串		
INS	串输入 (从I/O端口输入)		
OUTS	串输出 (向I/O端口输出)		

(1). MOVS 串传送

指令隐含格式: **MOVS DST, SRC**

显式: **MOVSB DST, SRC ; 字节传送**

MOVSW DST, SRC ; 字传送

MOVSD DST, SRC ; 双字传送

- 隐含格式: 应该在操作数中表明字节、字、双字传送操作

例如: **MOVS ES:BYTE PTR[DI], DS:[SI]**

- 显式格式: 经常使用, 根据指令操作符的最后一个字母决定字节、字、双字传送操作
- 不影响标志位

① 串处理指令中操作数

■ 寄存器型操作数：只能放在累加器中

- 字节操作数应放在AL中
- 字操作数放在AX中
- 双字操作数放在EAX中

■ 存储器型操作数：应先建立地址指针

- 源操作数时，必须把源串首地址偏移量放入SI寄存器（若地址长度是32位的则使用ESI），段缺省寻址DS所指向的段，允许使用段超越前缀
- 目标操作数时，必须把目标串首地址偏移量放入DI寄存器（若地址是32位的则使用EDI），段寄存器是ES

② 地址指针的修改

- 串指令执行后系统自动修改地址指针SI (ESI)、DI (EDI)
 - 字节型操作 ± 1
 - 字型操作 ± 2
 - 双字型操作 ± 4

③ 修改方向：基于方向标志DF

- $DF=0$ ，则地址指针增量
- $DF=1$ ，则地址指针减量
- 可以用CLD和STD指令对DF置0或1
- ◆ 串处理指令前应先建立地址指针、方向标志

(2) STOS指令

格式: **STOS DST; STOSB DST; STOSW DST; STOSD DST**

- 源操作数在AL、AX或EAX; 其他同MOVS
- 与REP结合使用在初始化某一缓冲区时很有用

(3) LODS指令

格式: **LODS SRC; LODSB SRC; LODSW SRC; LODSD SRC**

- 目的操作数在AL、AX或EAX; 其他同MOVS
- 将某一缓冲区数据逐个取出测试时很有用

(4) INS指令

格式: **INS** **DST, DX;** **INSB** **DST, DX;** **INSW** **DST, DX;** **INSD** **DST, DX**

- 源操作数在IO寄存器中, 其端口号 (IO寄存器地址) 在DX寄存器中
- 目标操作数、地址指针的修改、修改方向、标志位等同MOV指令规定
- 与REP前缀结合使用可以实现IO寄存器中连续数据送到存储缓冲区

(5) OUTS指令

格式: **OUTS** **DX, SRC;** **OUTSB** **DX, SRC;** **OUTSW** **DX, SRC;** **OUTS** **DX, SRC**

- 目标操作数地址是IO寄存器, 其端口号 (IO寄存器地址) 在DX寄存器中
- 源操作数、地址指针的修改、修改方向、标志位等同MOV指令规定
- 与REP前缀结合使用可以实现存储缓冲区的一组连续数据送到IO寄存器

- ◆ 操作数可以字节、字、双字
- ◆ 与REP前缀结合使用时注意执行速度和IO端口处理速度相匹配

(6) CMPS指令

格式: CMPS DST, SRC; CMPSB DST, SRC; CMPSW DST, SRC;
CMPSD DST, SRC

执行的操作:

两个字符串比较

- (SRC) - (DST), 但结果不保存, 根据结果设置标志位
- 目标操作数和源操作数的地址指针 (变址寄存器DI, SI) 的修改、修改方向同MOVS规定
- 操作数的规定和寻址方式: 同MOVS

(7) SCAS指令

格式: SCAS DST; SCASB DST; SCASW DST; SCASD DST

执行的操作:

累加器内容与字符串比较

- (AL/AX/EAX) - (DST), 但结果不保存, 根据结果设置标志位
- 目标操作数的地址指针 (变址寄存器DI) 的修改、修改方向同MOVS规定
- 目标操作数的规定和寻址方式: 同MOVS

3.3.5 重复前缀

- ◆ 单条的串指令只能处理数据串中的一个数据
 - ◆ 处理整个数据串时，必须要有重复前缀才可以
 - ◆ 串处理指令使用的前缀，必须与串指令一起使用
- MOVS、STOS、LODS、INS、OUTS指令前缀：

REP ; 重复

CMPS、SCAS指令前缀：

REPE/REPZ ; 相等/为零则重复

REPNE/REPNZ ; 不相等/不为零则重复

字符串比较和串中查找字符很有用

1. 重复 REP

格式: REP 串处理指令

- 串处理指令可以是 MOVS、STOS、LODS、INS、OUTS
- 例如: REP MOVES:BYTE PTR[DI], DS:[SI]

执行的操作:

- ① 如 (count reg)=0, 则退出REP, 否则, 往下执行
 - ② (count reg)-1 → (count reg)
 - ③ 执行其后的串处理指令
 - ④ 转①, 即重复①~③
- 16位地址时count reg是CX; 32位地址时count reg是ECX
 - 执行前必须设置好count reg

例如: REP MOVES:BYTE PTR[DI], DS:[SI]

- ① 如CX=0, 则退出REP, 该指令执行结束; 否则, 往下执行
- ② CX-1 → CX
- ③ 执行《MOVES:BYTE PTR[DI], DS:[SI]》串处理指令
 - (DS*16+SI) → (ES*16+DI)
 - DF=0时, SI+1 → SI, DI+1 → DI (字节操作, +1)
 - DF=1时, SI-1 → SI, DI-1 → DI
- ④ 转①, 即重复①~③

- ◆ **REP MOVSB ES:[DI], DS:[SI]** 指令相当如下一组指令：

```
RELOOP: CMP CX, 0  
        JZ NEXT  
        DEC CX  
        MOVSB ES:[DI], DS:[SI]  
        JMP RELOOP
```

```
NEXT:    .....
```

2. 相等重复前缀 REPE / REPZ

- 格式: REPE / REPZ CMPS或 SCAS指令

- 执行的操作为:

- ① 若 $CX=0$ (计数到) 或 $ZF=0$ (不相等), 则结束重复
- ② 否则, 修改计数器 $CX-1 \rightarrow CX$, 执行后跟的串操作指令。转①, 继续重复上述操作

- ◆ REPE CMPSB ES:[DI], DS:[SI]指令相当如下一组指令:

```
REPLoop: CMP CX, 0
          JZ NEXT
          DEC CX
          CMPSB ES:[DI], DS:[SI]
          JE REPLoop
```

```
NEXT:     .....
```

REPE CMPSB ES:[DI], DS:[SI]

; 比较, 并根据DF修改DI, SI

3. 不等重复前缀 REPNE / REPNZ

- 格式: REPNE / REPNZ CMPS或 SCAS指令

- 执行的操作为:

- ① 若 $CX=0$ (计数到) 或 $ZF=1$ (相等), 则结束重复
- ② 否则, 修改计数器 $CX-1 \rightarrow CX$, 执行后跟的串操作指令。转①, 继续重复上述操作

◆ REPNE CMPSB ES:[DI], DS:[SI]指令相当如下一组指令:

```
REPLoop: CMP CX, 0
          JZ NEXT
          DEC CX
          CMPSB ES:[DI], DS:[SI]
          JNE REPLoop
```

```
NEXT:     .....
```

REPNE CMPSB ES:[DI], DS:[SI]

;比较, 并根据DF修改DI, SI

对于串处理指令需要注意的其他几个问题：

1. 如果需要在同一段内传送或比较数据

- ① 在DS和ES中设置相同的段基地址
- ② 源操作数字段使用段跨越前缀

如 `MOVSB [DI], ES:[SI]`

2. count reg (CX) 中是重复执行次数

- 计数器：16位地址时用CX；32位地址时用ECX

3. 根据DF进行正向、反向传送或比较

- ◆ DF=0,操作数地址自动递增修改
- ◆ DF=1,操作数地址自动递减修改

例3.86

4. 执行前必须设置好count reg, DF, DI, SI

3.3.6 控制转移指令

控制转移指令包括：

1. 无条件转移指令
2. 条件转移指令
3. 条件设置指令
4. 循环指令
5. 子程序调用/返回
6. 中断指令/返回

1、无条件转移指令JMP

- | | |
|-------------|-------------------|
| (1) 段内直接短转移 | JMP SHORT OPR |
| (2) 段内直接近转移 | JMP NEAR PTR OPR |
| (3) 段内间接近转移 | JMP WORD PTR OPR |
| (4) 段间直接转移 | JMP FAR PTR OPR |
| (5) 段间间接转移 | JMP DWORD PTR OPR |

不影响标志位

(1) 段内直接短转移 `JMP SHORT OPR`

- $(IP_{\text{新}}) = (IP_{\text{原}}) + \text{8位位移量}$, CS不变
- 8位EA的位移量由目标地址OPR确定
- 80386及其后续机型: $(EIP_{\text{新}}) = (EIP_{\text{原}}) + \text{8位位移量}$, CS不变
- P102, 图3.22

```
JMP SHORT B1
A1:  ADD AX, BX
B1:  ...
```

(2) 段内直接近转移 `JMP NEAR PTR OPR`

- $(IP_{\text{新}}) = (IP_{\text{原}}) + \text{16位位移量}$, CS不变
- 16/32位目标指令EA的位移量由OPR确定
- 80386及其后续机型: $(EIP_{\text{新}}) = (EIP_{\text{原}}) + \text{32位位移量}$, CS不变

```
JMP NEAR PTR B2
A2:  ADD  AX, CX
      ...
B2:  SUB  AX, CX
```

(3) 段内间接近转移 `JMP WORD PTR OPR`

- EA值由OPR的寻址方式确定, 它可以使用除立即数方式以外的任何一种寻址方式
- $(IP_{\text{新}}) = (EA)$, CS不变
- 80386及其后续机型:
 $(EIP_{\text{新}}) = (EA)$, CS不变

```
LEA BX, B2 ; 装入有效地址
JMP WORD PTR BX
A2:  ADD AX, CX
      ...
B2:  SUB AX, CX
```

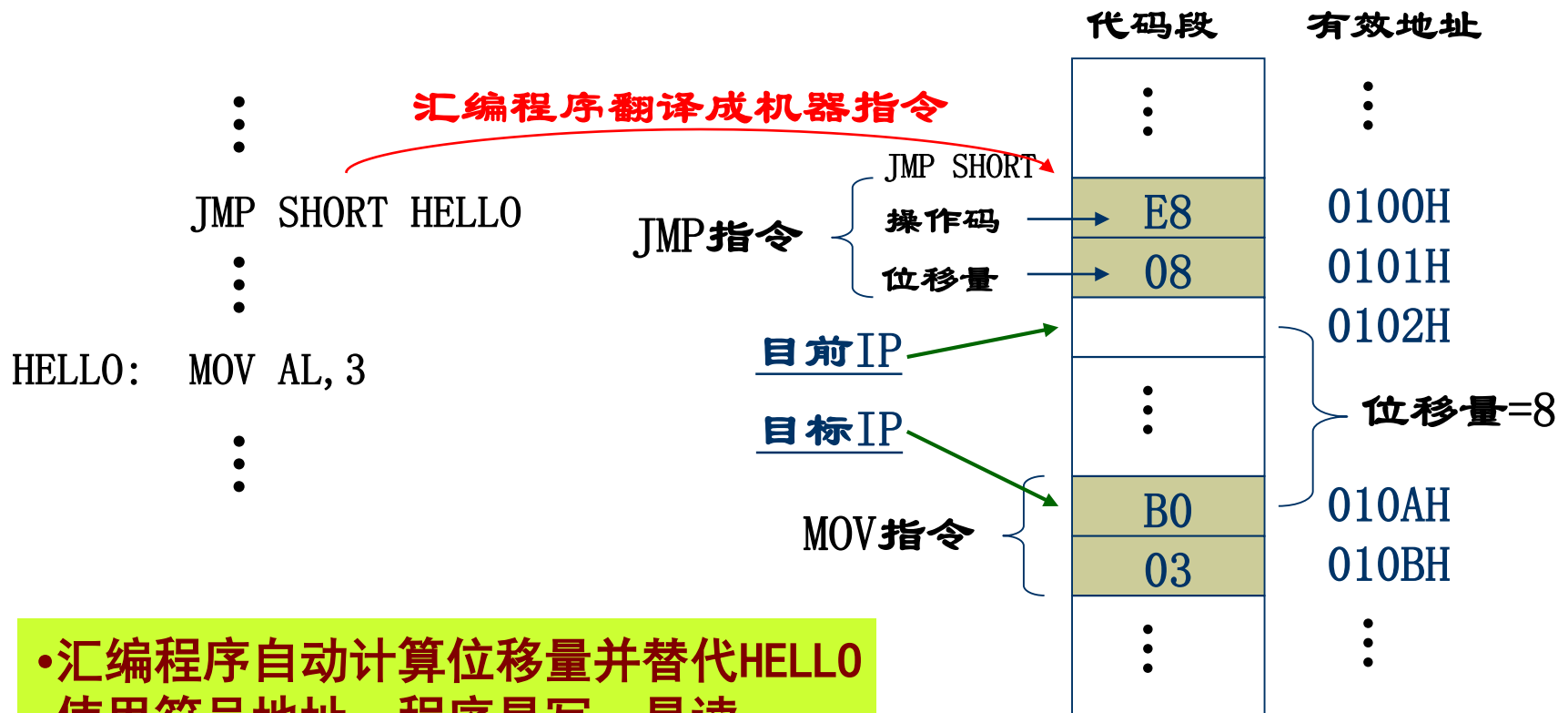
◆ 注意概念：

■ 偏移量=有效地址EA：

- 一般相对于段基地址（段开始地址）的位移量

■ 位移量：两个有效地址EA差， $EA_2 - EA_1$

- 位移量是带符号数



- 汇编程序自动计算位移量并替代HELLO
- 使用符号地址，程序易写、易读

(4) 段间直接转移 JMP FAR PTR OPR

OPR直接给出目标指令所在段的16/32位段基地址和段内有效地址EA

- ◆ $(IP_{\text{新}}) = \text{OPR给出的16位段内偏移量}$
- ◆ $(CS_{\text{新}}) = \text{OPR给出的16位段基地址}$

80386极其后续机型:

P103, 图3.23

- ◆ $(EIP_{\text{新}}) = \text{OPR给出的32位段内偏移量}$
- ◆ $(CS_{\text{新}}) = \text{OPR给出的32位段基地址}$

(5) 段间间接转移 JMP DWOR PTR OPR

OPR存储器寻址, 给出目标指令所在段的16/32位段基地址和段内有效地址EA

- ◆ $(IP_{\text{新}}) = \text{目标指令所在段的16位EA}$; 存储器间接寻址获得
- ◆ $(CS_{\text{新}}) = \text{目标指令所在段的16位段基地址}$; 存储器间接寻址获得

80386极其后续机型:

- ◆ $(EIP_{\text{新}}) = \text{目标指令所在段的32位EA}$; 存储器间接寻址获得
- ◆ $(CS_{\text{新}}) = \text{目标指令所在段的32位段基地址}$; 存储器间接寻址获得

C1 SEGMENT

⋮

JMP FAR PTR NEXT_PROG

⋮

C1 ENDS

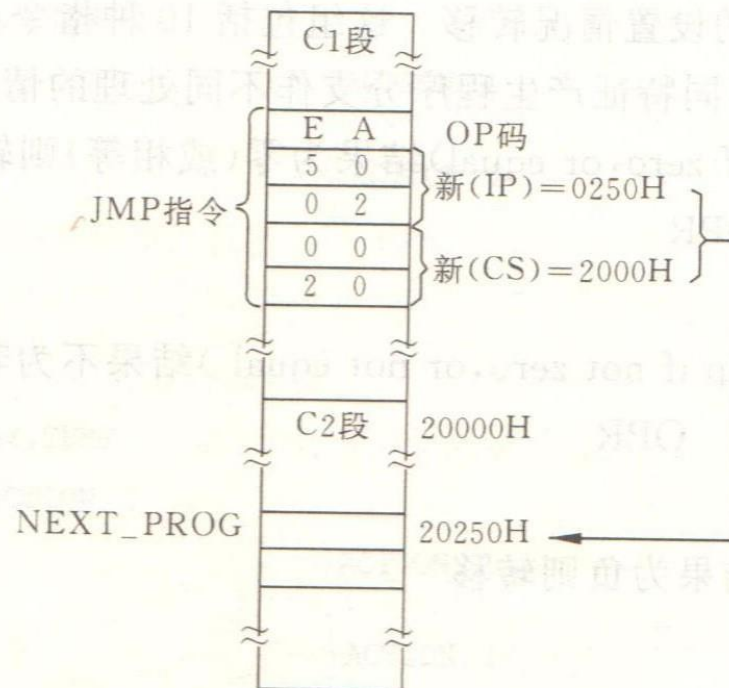
C2 SEGMENT

⋮

NEXT_PROG:

⋮

C2 ENDS



2、条件转移指令

指令格式 Jcc OPR

- OPR定义同无条件转移指令
- 根据上一条指令设置的条件码来判别测试条件
- 这类指令本身并不影响标志
- 分为4组来讨论

只能是段内短转移！

(1) 根据单个条件标志(状态位)的情况转移转移

JZ 结果为零转移

JNZ 结果不为零转移

JS 结果小于零转移

JNS 结果不小于零转移

JO 结果溢出转移

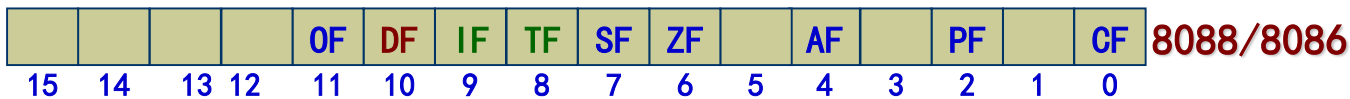
JNO 结果不溢出转移

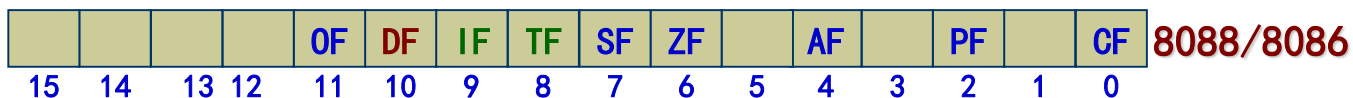
JP 结果中1的个数偶数转移

JNP 结果1的个数奇数转移

JC 结果进位/借位转移

JNC 结果没有进位/借位转移





{ JZ (或JE) 结果为零转移
JNZ (或JNE) 结果不为零转移

如果ZF=1, 转移
如果ZF=0, 转移

{ JS 结果小于零 (为负) 转移
JNS 结果不小于 (为正) 零转移

如果SF=1, 转移
如果SF=0, 转移

{ JO 结果溢出转移
JNO 结果不溢出转移

如果OF=1, 转移
如果OF=0, 转移

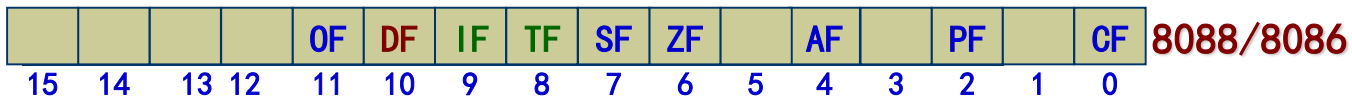
{ JP (或JPE) 结果1的个数偶数转移
JNP (或JPO) 结果1的个数奇数转移

如果PF=1, 转移
如果PF=0, 转移

{ JC (或JB, 或JNAE) 结果进位/借位转移
JNC (或JNB, 或JAE) 结果没有进位/借位转移

如果CF=1, 转移
如果CF=0, 转移

状态位由该指令前边的运算等指令决定, 这组指令只判断、跳转



(2) 比较两个无符号数，并根据比较结果转移*

A (高于), B (低于), E (等于)

格式

测试条件

<	JB (JNAE,JC)	OPR	CF = 1
≥	JNB (JAE,JNC)	OPR	CF = 0
≤	JBE (JNA)	OPR	CF∨ZF = 1
>	JNBE (JA)	OPR	CF∨ZF = 0

* 适用于地址或双精度数低位字的比较

指令JZ/JE和 JNZ/JNE同样可以用于两个无符号数的比较转移

比较两个无符号数由该指令的前边其他指令执行并设置条件标志，这组指令只判断、跳转

```
CMP DST, SRC
JB  NEXT
```

(3) 比较两个带符号数以后，根据比较结果转移 G (大于)，L (小于)，E (等于)

< JL (JNGE)

$SF \vee OF = 1$

\geq JNL (JGE)

$SF \vee OF = 0$

\leq JLE (JNG)

$(SF \vee OF) \vee ZF = 1$

> JNLE (JG)

$(SF \vee OF) \vee ZF = 0$

SF	OF	
0	0	0
0	1	1
1	0	1
1	1	0

SF	OF	ZF	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

比较两个无符号数由该指令的前边其他指令执行并设置条件标志，这组指令只判断、跳转

例：X、Y是带符号数，编制程序实现：如果 $X > 50$ ，转到 TOO_HIGH；否则 $|X - Y| \rightarrow \text{RESULT}$ ，如果溢出转到 OVERFLOW.

```
MOV    AX, X
CMP    AX, 50
JG     TOO_HIGH
SUB    AX, Y
JO     OVERFLOW
JNS    NONNEG
NEG    AX
```

NONNEG:

```
MOV    RESULT, AX
```

TOO_HIGH:

.....

OVERFLOW:

.....

(4) 测试CX或ECX的值为0则转移

- JCXZ CX寄存器内容为零则转移
 - JECXZ ECX寄存器内容为零则转移
-
- 这组指令只能提供8位位移量，即短转移

3. 条件设置指令

- ◆ 也称为条件字节设置指令，386及后继机型才提供这组指令
- ◆ 这些指令根据前边指令对状态标志位的设置，将**目标字节**置1或清0
 - 格式：SET_{cc} DST
 - 功能：如果条件为真，则置DST=1，否则DST=0
- ◆ 用于保存条件码，在以后适当时机再判断转移等
 - 这时用这组指令效率高、简单
- ◆ 这组指令不会影响标志位

指令格式	功 能	测试条件
SETZ/SETE DST	等于0/相等时置DST为1；否则，DST为0	ZF=1
SETNZ/SETNE DST	不等于0/不相等时置DST为1；否则，DST为0	ZF=0
SETS DST	为负时置DST为1；否则，DST为0	SF=1
SETNS DST	非负时置DST为1；否则，DST为0	SF=0
SETO DST	溢出时置DST为1；否则，DST为0	OF=1
SETNO DST	无溢出时置DST为1；否则，DST为0	OF=0
SETP/SETPE DST	结果低8位有偶数个1时置DST为1；否则，DST为0	PF=1
SETNP/SETPO DST	结果低8位有奇数个1时置DST为1；否则，DST为0	PF=0
SETG/SETNLE DST	大于/不小于等于时置DST为1；否则，DST为0	ZF=0 and SF=OF
SETNG/SETLE DST	不大于/小于等于时置DST为1；否则，DST为0	ZF=1 or SF≠OF
SETL/SETNGE DST	小于/不大于等于时置DST为1；否则，DST为0	SF≠OF
SETNL/SETGE DST	不小于/大于等于时置DST为1；否则，DST为0	SF=OF
SETA/SETNBE DST	高于/不低于等于时置DST为1；否则，DST为0	CF=0 and ZF=0
SETNA/SETBE DST	不高于/低于等于时置DST为1；否则，DST为0	CF=1 or ZF=1
SETB/SETNAE/SETC DST	低于/不高于等于/有进位时DST置1；否则，DST为0	CF=1
SETNB/SETAE/SETNC DST	不低于/高于等于/无进位时DST置1；否则，DST为0	CF=0

4. 循环指令

80X86为了简化循环程序的设计，设计了一组循环指令如下：

LOOP OPR

LOOPE/LOOPZ OPR

LOOPNE/LOOPNZ OPR

循环指令特点：

- ① 循环入口地址（指令中的LABEL）只能在当前IP值的
-128~+127范围之内， 即短转移
- ② 用CX或ECX（计数操作数长度为32位时）作为循环次数
的计数器
- ③ 不影响标志

思考题：循环入口地址, 为什么不能近转移或跨越段？

- 一般循环体比较小！一般情况下短转移指令就够用，指令系统越简单越好
- 循环指令在程序中出现和执行频率最高！循环指令应该短、快速

1. 循环指令 LOOP

格式：LOOP OPR

功能：

- ① $(CX) - 1 \rightarrow (CX)$
- ② 若 $(CX) \neq 0$ ，则转向标号处执行循环体，即 $IP + \text{位移量} \rightarrow IP$
 - 否则，退出循环体，顺序执行下一条指令，即 IP 不变

```
star: ...  
      ...  
      loop start  
      ...
```

2. 相等循环指令 LOOPE/LOOPZ

格式：LOOPE/LOOPZ OPR

功能：

- ① $(CX) - 1 \rightarrow (CX)$; 注意不影响ZF，这时ZF是前面指令的执行结果
- ② 若 $(CX) \neq 0$ and $ZF = 1$ ，则转向标号处执行循环体，
即 $IP + \text{位移量} \rightarrow IP$
 - 否则，退出循环体，顺序执行下一条指令，即 IP 不变

3. 不等循环指令 LOOPNE/LOOPNZ

格式: LOOPNE/LOOPNZ OPR

功能:

- ① $(CX) - 1 \rightarrow (CX)$;注意不影响ZF, 这时ZF是前面指令的执行结果
- ② 若 $(CX) \neq 0$ and $ZF = 0$, 则转向标号处执行循环体,
即 $IP + \text{位移量} \rightarrow IP$
 - 否则, 退出循环体, 顺序执行下一条指令,
即IP 不变

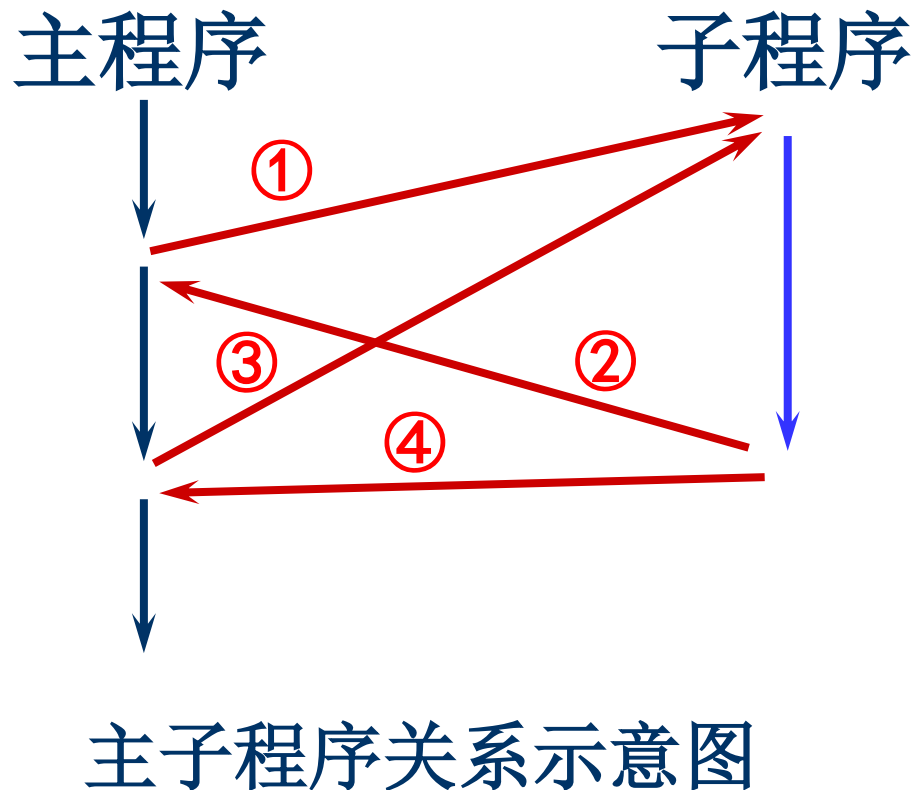
参看P113 例子3.96

5. 子程序调用/返回指令

- ◆ 子程序: 子程序结构相当于高级语言中的过程 (PROCEDURE). 为便于模块化程序设计, 往往把程序中某些具有独立功能的部分编写成独立的程序模块, 称之为子程序
- ◆ 程序中 由子程序调用指令调用子程序, 而在子程序执行完后由返回调用指令返回调用程序继续执行
- ◆ 80X86提供了以下指令
 - CALL 子程序调用
 - RET 子程序返回

汇编程序中一定要注意正确使用堆栈及返回方式等, 而高级语言不必关心这些

主程序和子程序关系



子程序多次被调用

1). CALL指令

(1) 段内直接近调用

CALL DST

(2) 段内间接近调用

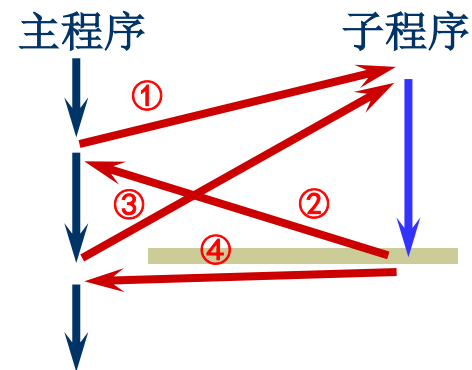
CALL DST

(3) 段间直接远调用

CALL DST

(4) 段间间接远调用

CALL DST



主子程序关系示意图

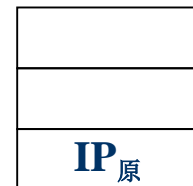
与无条件转移指令的转移方式相同，不同之处是CPU会自动将返回点(IP, CS)保存在堆栈中

① 段内直接调用：

格式：CALL DST

功能：执行时先把返回地址 ($IP_{原}$) 压入堆栈，
再使 $(IP_{新}) = (IP_{原}) + \text{指令中给出的16位位移量}$

堆栈



SP→

② 段内间接调用：

格式：CALL DST ; REG / M寻址方式

功能：执行时先把返回地址 ($IP_{原}$) 压入堆栈，
再使 $(IP_{新}) = \text{指令指定的16位通用寄存器或内存单元的内容}$

③ 段间直接调用：

格式：CALL DST

功能：执行时先把返回地址 (当前IP值和当前CS值) 压入堆栈，再把指令中给出的偏移量 (EA) 部分送给IP，段基址部分送给CS



SP→

④ 段间间接调用：

格式：CALL M

功能：执行时先把返回地址 (当前IP值和当前CS值) 压入堆栈，再把存储器中连续4字节的低字送给IP，高字送给CS

2) RET指令

段内返回	(1) 段内近返回	RET	
	(2) 段内带立即数近返回	RET	EXP
段间返回	(3) 段间远返回	RET	
	(4) 段间带立即数远返回	RET	EXP

- 执行这组指令可以返回到被调用处
- 不影响标志
- 返回地址隐含，地址在堆栈中，隐含执行下面操作：
 - ① 段内：POP IP(EIP)
 - ② 段间：POP IP(EIP), POP CS
- 带立即数的返回指令：主要是为了配合在调用子程序前通过堆栈给子程序传递参数

如何判断段内和段间返回？子程序段说明时会给出

(1) 段内回指令 RET

格式: RET

功能: 隐含执行 POP IP(EIP)

(2) 段间远返回 RET

格式: RET

功能: 隐含执行 ①POP IP(EIP); ②POP CS

(3) 段内带立即数的返回指令

格式: RET imm₁₆

功能: 隐含执行 ①POP IP(EIP); ②修改栈顶指针 (SP) + imm₁₆ → (SP)

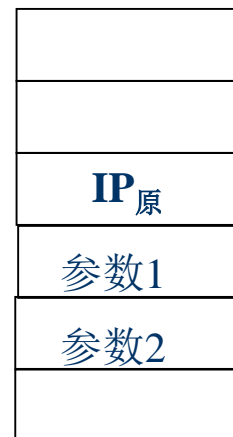
注: 其中imm₁₆是16位的立即数, 设通过堆栈给子程序传递了n个字型参数, 则imm₁₆ = 2n

(4) 段间带立即数远返回 RET EXP

格式: RET imm₁₆

功能: 隐含执行 ①POP IP(EIP); ②POP CS; ③修改栈顶指针 (SP) + imm₁₆ → (SP)

SP→



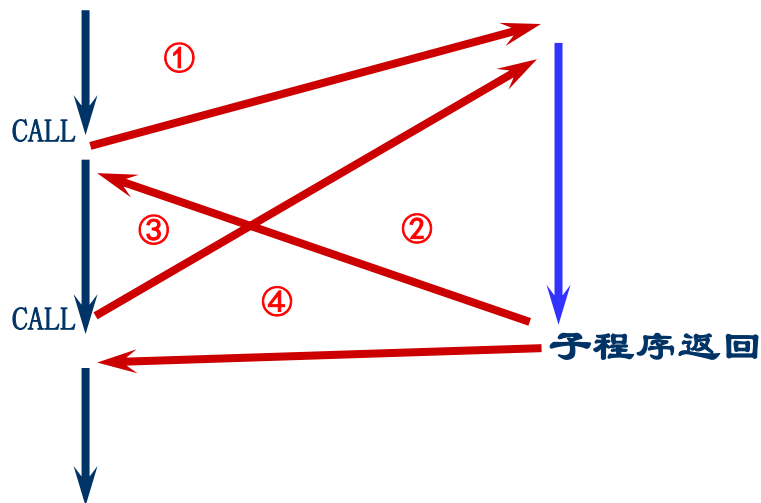
堆栈

6、 中断

- ◆ 有时当**系统**运行或者**程序**运行期间在遇到某些特殊情况时，需要计算机自动执行一组专门的程序来进行处理。这种情况称为中断，所执行的这组程序称为**中断例行程序或中断子程序**
- ◆ 其它随机事件，如I/O控制和数据传送，不采用中断方式系统效率会很低

主程序

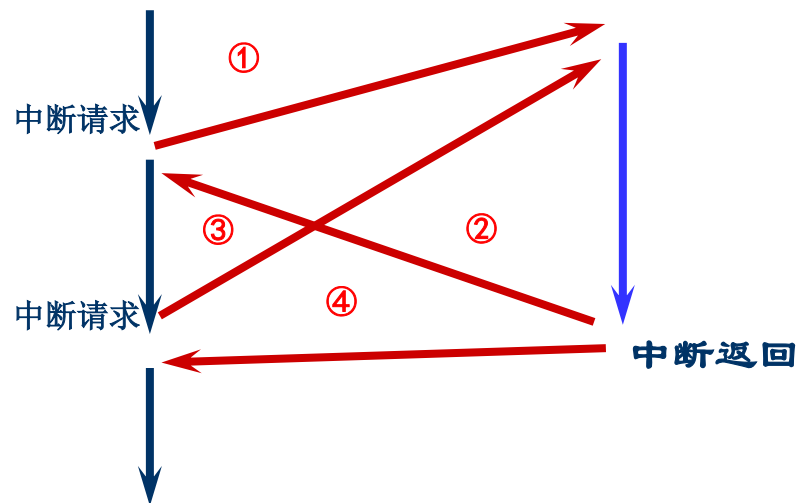
子程序



- 主程序调用
- 主、子程序都是程序的一部分

正在执行程序

中断子程序



- 中断请求打断正在执行程序
- 执行程序和中断程序关系无关

中断过程：向CPU发中断请求；CPU响应中断，中断正在执行程序，转中断子程序；返回被中断程序。

- CPU响应一次中断时，也要和调用子程序时类似地把指令指针和CS保存到堆栈。为了能全面的保存现场信息，以便在中断处理结束时返回现场，需要把反映现场状态的FLAGS保存入栈，然后才能转到中断服务程序去执行
- 从中断返回时，需要恢复指令指针IP、CS、FLAGS
- 有关中断的指令：（软中断）

INT	中断指令
INTO	如溢出则中断
IRET	从中断返回

中断请求方式：

1. 软中断：程序中安排
2. 硬件中断：与程序无关

1). 中断向量

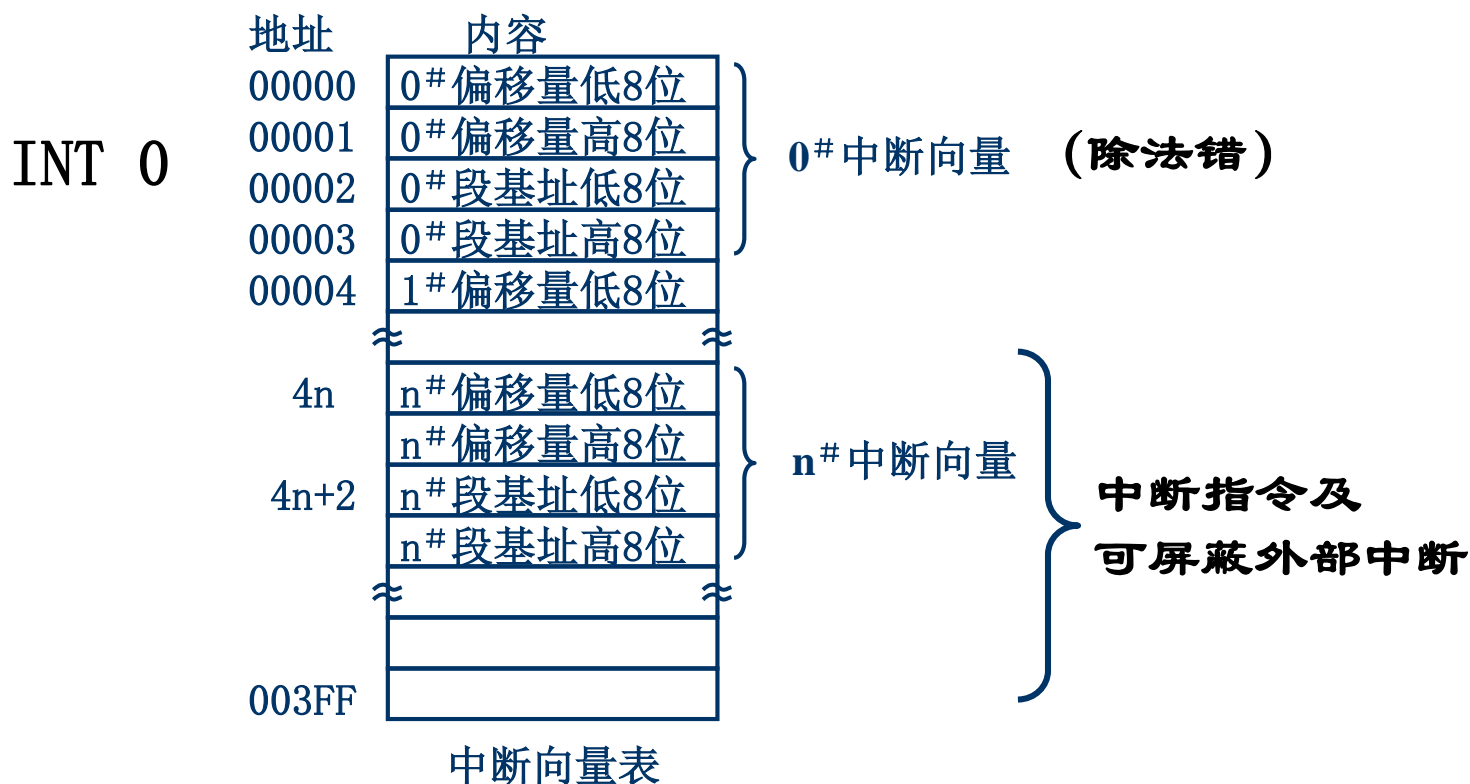
- 中断向量：中断处理子程序的入口地址
- 在PC机中规定中断处理子程序为FAR型
 - 每个中断向量占用4个字节，其中低两个字节为中断向量的偏移量部分，高两个字节为中断向量的段基址部分

2). 中断类型号

- IBM PC机共支持256种中断，相应编号为0~255，把这些编号称为中断类型号

3). 中断向量表

- ◆ 256种中断有256个中断向量，把这些中断向量按照中断类型号由小到大的顺序排列，形成中断向量表
- ◆ 表长为 $4 \times 256 = 1024$ 字节，从内存的0000:0000地址开始存放，占用内存最低端的0~3FFH单元
- ◆ 请见图3. 29



4). 软件中断指令 INT

- 在8086 / 8088中, 中断分为内中断(或称软中断)和外中断(或称硬中断), 这里只介绍内中断的中断调用指令
- 格式: `INT n` ; n 为中断类型号
- 功能: 中断当前正在执行的程序, 把当前的FLAGS、CS、IP值依次压入堆栈(保护断点), 关中断 ($IF=0$)、陷阱 ($TF=0$) 和对齐 ($AC=0$), 然后从中断向量表的 $4n$ 处取出 n 类中断向量, 其中 $(4n) \rightarrow IP$, $(4n+2) \rightarrow CS$, 转去执行中断处理子程序

5). 中断返回指令 IRET

- 格式: `IRET`
- 功能: 从栈顶弹出三个字分别送入IP、CS、FLAGS寄存器 (按中断调用时的逆序恢复断点), 返回到原程序断点继续执行

6). 子程序和中断子程序中特别注意：

- ◆ 为了保证返回地址正确，子程序中PUSH和POP必须一对一出现；
- ◆ 凡子程序中使用寄存器，必须在子程序开始推入堆栈，子程序返回指令前出栈以**按序**恢复寄存器内容；
 - 有些CPU机型在中断响应/返回时另外也自动保存/恢复了常用寄存器
- ◆ 注意关中断和开中断时机！

3.3.7 处理器控制指令

1. 标志处理指令

汇编格式	功 能	影响标志
CLC (Clear Carry)	把进位标志CF清0	CF
STC (Set Carry)	把进位标志CF置1	CF
CMC (Complement Carry)	把进位标志CF取反	CF
CLD (Clear Direction)	把方向标志DF清0	DF
STD (Set Direction)	把方向标志DF置1	DF
CLI (Clear Interrupt)	把中断允许标志IF清0	IF
STI (Set Interrupt)	把中断允许标志IF置1	IF

2. 其他处理机控制指令

不影响标志

名称	汇编格式	功 能	说 明
空操作	NOP (No Operation)	空操作	CPU不执行任何操作, 其机器码占用一字节
停机	HLT (Halt)	使CPU处于 停机状态	只有外中断或复位信号 才能退出停机, 继续执行
等待	WAIT (Wait)	使CPU处于 等待状态	等待TEST信号有效后, 方可退出等待状态, 继续执行
锁定 前缀	LOCK (Lock)	使总线锁定 信号有效	LOCK是一个单字节前缀, 在其后的 指令执行期间, 维持总线的锁 存信号直至该指令执行结束

如何查阅

《附录1 80x86指令系统一览表》

助记符	汇编语言格式		功 能	操作数	时钟 周期数	字节数	标志位										备注
							O	D	I	T	S	Z	A	P	C		
AAA	AAA		AL←把 AL 中的和调整到非压缩的 BCD 格式		3	1	u	-	-	-	u	u	x	u	x		
ADD	ADD	dst, src	(dst) ← (dst) + (src)	reg, reg	1	2	x	-	-	-	x	x	x	x	x		
				reg, mem	2	2~7											
				mem, reg	3	2~7											
				reg, imm	1	3~6											
				ac, imm	1	2~5											
				mem, imm	3	3~11											
JCXZ	JCXZ	opr	CX=0 则转移		6/5	2	-	-	-	-	-	-	-	-	-		
RET	RET		段内: IP ← POP ()		2	1	-	-	-	-	-	-	-	-	-		
			段间: IP ← POP () CS ← POP ()		4*	1											
IRET	IRET		IP ← POP () CS ← POP () FLAGS ← POP ()		7*	1	r	r	r	r	r	r	r	r	r		

* 实模式下的时钟周期数。在保护模式下，由于情况复杂，未提供

习 题

3. 1

3. 2

3. 3

3. 4

3. 11

3. 12

3. 13

3. 14

3. 17

3. 30

3. 32

3. 55

3. 66